UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Avaliação do uso de agrupamento de dados de desempenho para apoiar o teste de software no domínio de aprendizagem de máquina

Diego Braga

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP SÃO CARLOS

Diego Braga

Avaliação do uso de agrupamento de dados de desempenho para apoiar o teste de software no domínio de aprendizagem de máquina

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Paulo Sergio Lopes de Souza

USP – São Carlos
Novembro de 2022

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

Braga, Diego
BB813a Avaliação do uso de agrupamento de dados de desempenho para apoiar o teste de software no domínio de aprendizagem de máquina / Diego Braga; orientador Paulo Sergio Lopes de Souza. -- São Carlos, 2022.
125 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2022.

1. Aprendizagem de máquina. 2. Teste de software. 3. Agrupamento de dados. 4. Monitoração de desempenho. I. de Souza, Paulo Sergio Lopes, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Diego Braga

Assessing the use of performance data clustering for supporting software testing in the machine learning domain

Dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Master in Science. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Paulo Sergio Lopes de Souza

USP – São Carlos
November 2022

À Alessandra, por todo o apoio e parceria.

AGRADECIMENTOS

Agradeço aos meus orientadores, Prof. Dr. Paulo Sergio Lopes de Souza e Profa. Dra. Simone Do Rocio Senger de Souza, pela oportunidade, atenção, paciência e aprendizado.

Agradeço ao Thiago, Vitor e demais colegas do ICMC, pelas ideias, suporte e companheirismo.

Finalmente, agradeço a minha esposa Alessandra, por toda a ajuda e paciência durante essa jornada.

“O sucesso é ir de fracasso em fracasso sem perder o entusiasmo.”
(Winston Churchill)

RESUMO

BRAGA, D. **Avaliação do uso de agrupamento de dados de desempenho para apoiar o teste de software no domínio de aprendizagem de máquina**. 2022. 125 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2022.

Inteligência Artificial (IA) é um dos campos mais novos da ciência e engenharia. A grande quantidade de dados disponíveis tornou possível o desenvolvimento da Aprendizagem de Máquina (AM), disciplina de IA que é capaz de criar aplicações com a habilidade de se otimizar a partir da análise de dados. Sistemas de AM têm sido utilizados em cada vez mais domínios, mas apesar de sua crescente popularidade, a garantia de qualidade neste campo de IA ainda é um desafio. Para se buscar a qualidade de aplicações de AM são necessárias abordagens de Verificação, Validação e Teste (VV&T) de software capazes de detectar defeitos nesses tipos de software. Testes em sistemas de software de IA tendem a ser desafiadores pois estas aplicações geram resultados difíceis de se prever, alcançados a partir de algoritmos e modelos de aprendizado criados pelos próprios sistemas de software de AM. Dadas essas características, o uso de técnicas de teste que tratam tais sistemas de software como caixas-pretas e não precisam verificar suas saídas, tornam-se bem atrativas. A metodologia Tricorder demonstra ser promissora neste domínio como uma técnica de teste complementar às técnicas de teste já conhecidas, permitindo estender os testes aplicados sem que sejam necessários acessos ao código-fonte ou a criação de casos de teste tradicionais. A metodologia Tricorder utiliza dados de monitoramento para definir um perfil de desempenho da aplicação sendo monitorada. Ao criar o perfil de desempenho de uma nova versão desta mesma aplicação, a Tricorder é capaz de comparar os dois e acusar possíveis defeitos com base no uso de recursos do sistema computacional. Estes perfis de desempenho são criados com o agrupamento de dados de desempenho monitorados, seguindo a metodologia DAMICORE. Apesar de promissora, a metodologia Tricorder ainda se encontra em fase de desenvolvimento e validação, tendo sido proposta apenas nos últimos anos. O estudo reportado nesta dissertação analisa a aplicação da metodologia Tricorder em sistemas de AM. Este estudo verifica se Tricorder pode ser empregada como uma abordagem complementar de teste à detecção de defeitos inseridos em aplicações deste ramo de IA. Resultados dos experimentos demonstram que a Tricorder detectou automaticamente defeitos que não afetam as saídas das aplicações selecionadas, demonstrando ser eficaz nas condições definidas neste estudo. Estes resultados contribuem para o estado da arte de VV&T de aplicações de AM, por permitir que uma nova metodologia de teste complementar e automatizada possa ser empregada em um futuro próximo.

Palavras-chave: Aprendizagem de máquina, Teste de software, Agrupamento de dados, Monitoração de desempenho.

ABSTRACT

BRAGA, D. **Assessing the use of performance data clustering for supporting software testing in the machine learning domain**. 2022. 125 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2022.

Artificial Intelligence (AI) is one of the newest fields in science and engineering. Large amounts of available data made it possible to develop Machine Learning (ML), the AI discipline capable of creating applications that can optimize themselves through data analysis. Due to their capabilities, ML software has been used in more and more domains, but quality assurance in this field of AI still is a challenge despite its growing popularity. For quality assurance of ML applications, it is necessary to find software Verification, Validation, and Test (VV&T) approaches capable of detecting defects in this type of software. Tests in AI software tend to be challenging because these applications have the characteristic of generating results that are difficult to predict, calculated by algorithms and learning models created by the ML software itself. Due to these characteristics, the use of test techniques that treat these software as black boxes and do not require checking their outputs, become very attractive. The Tricorder methodology is promising as a complementary technique for already known software testing approaches, allowing the extension of tests without requiring access to the source code or creating traditional test cases. The Tricorder methodology uses monitoring data to define a performance profile of the application being monitored. When creating the performance profile of a new version of the same application, Tricorder can compare the two and point out possible defects based on the resource usage of the computational system. These performance profiles are created by grouping monitored performance data, following the DAMICORE methodology. Although promising, the Tricorder methodology is still under development and validation, having been proposed only in recent years. This study analyzes the behavior of the Tricorder methodology when applied in systems based in ML, generating data that shows if it can be employed as a complementary approach for the detection of defects in applications of this AI field. The results obtained demonstrate that the methodology was able to achieve great results in the conditions stablished in this study, being able to detect defects that do not affect the outputs of the selected projects. These results contribute to the state of the art of VV&T of ML applications, by proposing a new, complementary and automatic methodology that could be employed in a close future.

Keywords: Machine learning, Software testing, Data grouping, Performance monitoring.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

CAPÍTULO

1

INTRODUÇÃO

1.1 Contexto

A Inteligência Artificial (IA) é um dos campos mais novos da ciência e engenharia, tendo sido mencionada pela primeira vez em 1956 (RUSSELL; NORVIG, 2009). Somente na última década sistemas de IA foram capazes de atingir sua proposta inicial, a de realizar tarefas até então consideradas apenas possíveis para humanos e nunca para computadores (DOšILOVIć; BRčIć; HLUPIć, 2018).

Aplicações de IA mais sofisticadas, que podem até mesmo atingir um desempenho equiparável ao humano em algumas atividades, se tornaram realidade devido ao cumprimento de dois critérios essenciais: o aumento do poder computacional e, principalmente, o crescimento do volume de dados disponíveis (DOšILOVIć; BRčIć; HLUPIć, 2018). No mundo atual, dados têm recebido cada vez mais importância, de tal modo que a população em geral tem passado a se preocupar com questões como coleta e vazamento de dados. Por esses motivos é possível até mesmo afirmar que esta é a era dos dados (ALPAYDIN, 2014).

A grande quantidade de dados disponíveis tornou possível o desenvolvimento de uma disciplina da IA, chamada de Aprendizagem de Máquina, que é capaz de criar aplicações que conseguem se otimizar a partir da análise de dados. A área de Aprendizagem de Máquina utiliza dados para possibilitar a resolução de problemas que não são facilmente resolvidos por raciocínio lógico tradicional (ALPAYDIN, 2014) (FLACH, 2012). Em outras palavras, é a área de estudo que permite que computadores aprendam sem que sejam explicitamente programados para isso (SAMUEL, 1959). Este estudo tem como foco o campo de Aprendizagem de Máquina.

Já faz alguns anos que a Aprendizagem de Máquina é utilizada para resolver problemas associados a uma grande variedade de domínios, como classificação de textos, processamento de linguagens naturais, reconhecimento de fala, reconhecimento de imagens, detecção de fraudes, jogos e diagnósticos médicos (RUSSELL; NORVIG, 2009). Com a popularização deste ramo de

IA, é esperado que seu uso se amplie para cada vez mais campos (MOHRI; ROSTAMIZADEH; TALWALKAR, 2012).

Apesar da crescente popularidade de aplicações de Aprendizagem de Máquina, a garantia de qualidade neste campo de IA ainda pode ser considerada como um desafio. Na literatura ainda existem poucos recursos que auxiliam na qualidade deste tipo de software, como padrões, técnicas de testes e boas práticas de desenvolvimento. Apesar de sistemas de software de Aprendizagem de Máquina serem capazes de usar experiências passadas para aprender a evitar novos problemas, seus algoritmos ainda estão sujeitos ao erro humano, assim como nos demais tipos de algoritmos (KIRK, 2014).

Para se garantir a qualidade de aplicações de Aprendizagem de Máquina, é necessário encontrar abordagens de Verificação, Validação e Teste (VV&T) de software que possam detectar defeitos e checar se estes programas realizam o que foram desenvolvidos para fazer: aprender (MYERS; SANDLER; BADGETT, 2011). As atividades de VV&T têm papel essencial no desenvolvimento de qualquer tipo de aplicação, pois permitem identificar e eliminar falhas que possam prejudicar o funcionamento de um software (SOUZA *et al.*, 2000).

A realização de testes em software de IA tende a ser especialmente desafiadora, pois os resultados destas aplicações, em específico aplicações de Aprendizagem de Máquina, têm a característica de não serem únicos. Aplicações de Aprendizagem são capazes de criar algoritmos e modelos de aprendizados cujos resultados são desconhecidos para o testador, fazendo com que, em muitos casos, não seja possível saber facilmente (analisando rapidamente apenas sua saída) se o resultado está correto.

Devido às características de Aprendizagem de Máquina é comum o uso de testes que tratam o software como uma caixa-preta, como técnicas de teste funcional (KIRK, 2014). Técnicas de teste que necessitam de acesso ao código-fonte, como é o caso da técnica de teste estrutural, não encontram tanto êxito neste contexto pois nele é comum o uso de bibliotecas e ferramentas especializadas, tornando mais difícil a verificação detalhada do fluxo de execução.

Apesar da dificuldade em testar sistemas de software de Aprendizagem de Máquina, a crescente popularidade deste tipo de aplicação e o seu uso em cada vez mais domínios, faz com que seja necessária a criação de novas formas de teste que sejam capazes de complementar o processo de garantia de qualidade nas aplicações deste ramo de IA.

Considerando as peculiaridades de Aprendizagem de Máquina, a metodologia de teste de software chamada Tricorder, proposta por Montes (MONTES, 2019) se demonstra promissora, pois permite a realização de testes sem que seja necessário acesso ao código-fonte do sistema sob teste, e não depende da criação de casos de teste. Esta metodologia possibilita a detecção de defeitos a partir do monitoramento de dados de desempenho, como por exemplo, o uso de processador, memória principal e acessos de entrada e saída.

A proposta da Tricorder é complementar às técnicas de teste (estrutural, funcional,

baseada em defeitos e em modelos) e não visa à substituição das mesmas (MYERS; SANDLER; BADGETT, 2011). Espera-se que tais técnicas continuem sendo usadas e possam revelar defeitos em testes unitários, de integração e de sistema (MYERS; SANDLER; BADGETT, 2011). A Tricorder complementa essas técnicas, sendo aplicada principalmente quando os sistemas já foram liberados para uso e defeitos não revelados ainda persistem no código. Outro uso peculiar da Tricorder é na VV&T de alterações do código em virtude de manutenções do sistema.

A metodologia Tricorder utiliza dados de monitoramento para definir o perfil de desempenho da aplicação sendo monitorada e ao criar o perfil de desempenho de uma nova versão da aplicação, é capaz de comparar os dois perfis e acusar possíveis defeitos, assumindo, como heurística, que o novo perfil não é o esperado para uma execução correta da aplicação. A comparação de perfis busca encontrar um comportamento atípico presente em somente um dos perfis, o qual pode ser causado por um defeito de software. Desta forma a metodologia se demonstra especialmente interessante para a fase de manutenção do ciclo de vida do software, onde é possível comparar o perfil de uma versão assumida como correta e uma nova versão gerada a partir de uma manutenção ou nova implementação.

Os perfis de desempenho são criados com o agrupamento dos dados de monitoramento, realizado seguindo a metodologia DAMICORE (SANCHES; CARDOSO; DELBEM, 2011). Esta abordagem de agrupamento utiliza a técnica *Normalized Compression Distance* (NCD), que calcula a distância entre os dados com o uso de compressão binária. O uso de NCD torna a metodologia Tricorder bastante flexível pois como os dados são trabalhados em sua estrutura binária, esta técnica é capaz de utilizar qualquer tipo de dado, permitindo o uso de diferentes métricas de desempenho e até mesmo diferentes ferramentas de monitoramento.

Por ser capaz de detectar defeitos sem precisar analisar as saídas ou acessar o código, a metodologia Tricorder mostra-se promissora para a realização de testes em aplicações de Aprendizagem de Máquina, não sendo impactada pelas características de IA que tornam difícil a garantia de qualidade destas aplicações.

Apesar de promissora, a metodologia Tricorder se encontra em fase de validação, tendo sido proposta apenas nos últimos anos (MONTES, 2019). No trabalho original a metodologia foi validada aplicando-a em um sistema do tipo cliente-servidor. Para a realização de seus experimentos, Montes (MONTES, 2019) criou variações desse sistema, cada uma com a injeção de um tipo diferente de falha e então aplicou a metodologia Tricorder comparando o sistema original com cada uma das versões com falhas.

Este estudo estende o trabalho original de Montes (MONTES, 2019) e verifica o desempenho da metodologia Tricorder em aplicações de Aprendizagem de Máquina, gerando dados que demonstram que a metodologia pode ser uma opção válida para complementar a realização de testes neste contexto e, em paralelo, colabora com o desenvolvimento da Tricorder, promovendo que a metodologia seja usada amplamente pela comunidade de teste de software.

1.2 Problema

Os experimentos apresentados por Montes (MONTES, 2019) demonstram que a metodologia foi eficaz na detecção das falhas injetadas. Porém, devido ao fato do experimento ter sido realizado em ambiente controlado, com uma aplicação e falhas desenvolvidas em um contexto específico de aplicação, a metodologia ainda possui algumas ameaças à sua validade em outros contextos.

É necessário realizar mais testes com a metodologia Tricorder para verificar a abrangência do uso do desempenho para a detecção de defeitos como uma abordagem de teste de software válida. Neste momento se faz necessário também aplicar a metodologia em diferentes tipos de aplicação, como em Aprendizado de Máquina, para demonstrar que ela pode ser eficaz na detecção de defeitos em software de maneira mais abrangente.

1.3 Motivação

O teste de software é uma atividade crucial para o desenvolvimento de aplicações, independentemente do domínio no qual está sendo aplicado. Apesar disso, o processo de VV&T ainda dispõe de um número limitado de técnicas e ferramentas que possam ser usadas para detectar defeitos em aplicações de Aprendizagem de Máquina.

A crescente popularidade da área de Aprendizagem de Máquina faz com que a detecção de defeitos em suas aplicações se torne uma necessidade cada vez maior, sendo necessária a criação de novas abordagens de teste que sejam capazes de promover a qualidade destas aplicações.

Este estudo foi proposto devido à necessidade de se avaliar a aplicabilidade da metodologia de teste Tricorder, proposta por Montes (MONTES, 2019), para a detecção de defeitos em aplicações de Aprendizagem de Máquina. Por meio deste objetivo espera-se também contribuir para a área de VV&T, especialmente no que diz respeito à qualidade de aplicações de Aprendizagem de Máquina.

1.4 Hipótese

A metodologia Tricorder apresenta características que a tornam promissora para o teste de aplicações de Aprendizagem de Máquina, como uma abordagem complementar às técnicas de teste já existentes, devido à possibilidade de remoção da necessidade de acesso ao código-fonte ou à criação de casos de teste. Tais características fazem com que a Tricorder possa ser um complemento de baixo custo ao teste de software, exigindo pouco investimento de tempo para sua configuração e podendo ser usada de forma automática, não sendo necessário o acompanhamento de um analista.

A hipótese seguida neste estudo foi a de que, devido às características já apresentadas, a metodologia Tricorder é capaz de encontrar defeitos em sistemas de software de Aprendizagem de Máquina, com um investimento pequeno e sendo eficaz. Com isso, a metodologia Tricorder poderia ser usada na fase de manutenção de software e de forma complementar aos testes de regressão já conhecidos na literatura de teste de software. Além disso, considerando o contexto específico de Aprendizagem de Máquina, do inglês *Machine Learning* (ML), a metodologia poderia ser utilizada também em um fluxo de *Machine Learning Operations* (MLOps), detalhado posteriormente neste trabalho.

1.5 Objetivo

Este estudo tem como objetivo verificar o comportamento da metodologia Tricorder quando utilizada em aplicações de Aprendizagem de Máquina, isto é, um contexto diferente do apresentado no trabalho original que a propôs (MONTES, 2019). Para verificar a eficácia citada, são utilizados os acertos e os números de falsos positivos e falsos negativos que ocorrem na detecção de defeitos ao usar a Tricorder nestes tipos de aplicações.

Para alcançar os objetivos deste trabalho foram investigadas novas métricas e ferramentas, assim como realizados novos experimentos, de forma a caracterizar a eficácia da metodologia no contexto de Aprendizagem de Máquina.

1.6 Contribuições Esperadas

Com a realização deste estudo é esperado que os resultados encontrados demonstrem que a Tricorder pode ser eficaz na detecção de defeitos em aplicações de Aprendizagem de Máquina e que é capaz de apresentar uma relação custo-benefício boa de forma que possa ser utilizada como uma técnica complementar de teste de software.

Espera-se contribuir para o estudo de garantia de qualidade de aplicações de Aprendizagem de Máquina, demonstrando que a metodologia é uma opção de baixo investimento, capaz de detectar defeitos desconhecidos. Com isso, espera-se que este trabalho possa ser usado como referência, contribuindo para o desenvolvimento da área de VV&T em geral e permitindo que a metodologia seja alvo de novos estudos e possa ser utilizada de maneira ampla por desenvolvedores de sistemas e profissionais de teste de software.

1.7 Estrutura

Neste primeiro capítulo foram apresentados o contexto, problemas e lacunas, motivações, hipóteses, e objetivos deste trabalho. A seguir é apresentada uma visão geral dos demais capítulos que compõem este documento.

No Capítulo 2 são apresentadas as fundamentações teóricas utilizadas ao decorrer deste estudo: Aprendizagem de Máquina, Validação, Verificação e Teste de Software e Monitoração de Sistemas Computacionais. No Capítulo 3 é apresentado o levantamento bibliográfico realizado para este trabalho e são detalhados os trabalhos relacionados identificados. No Capítulo 4 é apresentada em detalhes a metodologia Tricorder, o estado atual do seu desenvolvimento e os experimentos realizados por Montes (MONTES, 2019) para realizar uma validação inicial da metodologia.

No Capítulo 5 é apresentado o planejamento realizado para a execução dos experimentos. No Capítulo 6 são apresentados os resultados obtidos e a discussão sobre estes. Por fim, no Capítulo 7 são apresentadas as conclusões deste trabalho.

CAPÍTULO

2

FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações Iniciais

Este capítulo apresenta conceitos, terminologias e outras informações relevantes relacionadas aos principais temas utilizados neste trabalho. O capítulo é dividido em três sessões, cobrindo cada um dos temas: Aprendizagem de Máquina, VV&T, e por último, Monitoração de Sistemas Computacionais.

2.2 Aplicações de Aprendizagem de Máquina

O campo de IA, do inglês *Artificial Intelligence*, tem como objetivo entender o que são e então construir entidades inteligentes (RUSSELL; NORVIG, 2009)). Em termos práticos, aplicações de inteligência artificial são sistemas de software que utilizam algoritmos não-numéricos para solucionar problemas complexos que muitas vezes não podem ser solucionados pela computação tradicional ou por um raciocínio linear (PRESSMAN, 2009).

A área de IA é composta por múltiplas disciplinas, dentre as quais se destaca a Aprendizagem de Máquina. Esta área tem como foco o uso de métodos computacionais para utilizar informações disponíveis para a aplicação para aumentar o seu desempenho e possibilitar a realização de tarefas ou predições (MOHRI; ROSTAMIZADEH; TALWALKAR, 2012).

Aplicações de IA e principalmente de Aprendizagem de Máquina são partes essenciais deste estudo, sendo usadas como parte da metodologia Tricorder e também como objeto de teste. Nesta seção são apresentados os principais conceitos de software de Inteligência Artificial e Aprendizagem de Máquina que serão utilizados ao decorrer deste trabalho.

2.2.1 Definições básicas de IA

Desde a sua primeira aparição até os dias atuais, o termo Inteligência Artificial foi definido e estudado na literatura de diferentes formas, seguindo basicamente quatro aspectos compostos de duas dimensões de pensamento, conforme demonstrado na Figura 1. A primeira dimensão é dividida em duas filosofias, a dos que entendem que IA deve **pensar**, considerando um processo de raciocínio, e a dos que acreditam que IA deve **agir**, executando tarefas de acordo com um raciocínio. A segunda dimensão acredita em duas definições distintas, a de que aplicações IA devem ser fiéis ao comportamento **humano**, e a de que sistemas de software de IA devem ser **racionais**, buscando um desempenho ideal (RUSSELL; NORVIG, 2009).

Figura 1 – Categorias de IA

Pensamento Humano	Pensamento Racional
Ação Humana	Ação Racional

Fonte: Adaptada de Russell e Norvig (2009).

As duas dimensões de entendimento sobre IA podem ser sobrepostas, formando quatro categorias de definição:

- **Pensamento humano**: IA é o esforço para criar máquinas que pensam, máquinas com mentes (HAUGELAND, 1985). A automação de atividades associadas ao pensamento humano, como tomar decisões, resolver problemas e aprender (BELLMAN, 1978);
- **Pensamento racional**: IA é o estudo das faculdades mentais, utilizando modelos computacionais (CHARNIAK; MCDERMOTT, 1986). É o estudo das computações que permitem perceber, ter razão e agir (WINSTON, 1992);
- **Ação humana**: IA é a arte de criar máquinas que conseguem executar atividades que necessitam de uma inteligência humana (KURZWEIL, 1990). O estudo de como fazer computadores realizarem tarefas que, até o momento, humanos são melhores (RICH; KNIGHT, 1990);
- **Ação racional**: IA é o estudo de desenvolver agentes inteligentes (POOLE; MACKWORTH; GOEBEL, 1998). Tem como foco o comportamento inteligente de artefatos (NILSSON; NILSSON, 1998).

A proposta de ação humana segue os conceitos apresentados pelo **teste de Turing**, o qual propõe operações que possam definir de forma satisfatória o que é inteligência. Em sua essência, esse teste segue a ideia de que ao questionar uma máquina inteligente e receber suas respostas, um humano não seria capaz de distinguir se as respostas foram criadas por uma pessoa ou uma máquina.

Para que uma máquina seja capaz de passar no teste de Turing, ela precisaria das seguintes capacidades: **processamento de linguagens naturais**, para conseguir se comunicar, **representação de conhecimento**, para armazenar as informações transmitidas na questão, **raciocínio automatizado**, para armazenar as informações necessárias para responder a questão e chegar a novas conclusões e **aprendizado de máquina** para se adaptar a novas circunstâncias e reconhecer e extrapolar padrões (RUSSELL; NORVIG, 2009).

As capacidades necessárias consideram a proposta original de Turing, onde não haveria interação física direta entre o humano e a máquina. Por este motivo foi proposto o **teste total de Turing**, que precisa incluir sinais de vídeo para que as capacidades de percepção da máquina sejam testadas e também habilidades para que a máquina seja capaz de receber objetivos físicos, caso esta forma de troca de informação seja necessária. Assim, mais duas capacidades foram definidas: **visão computacional**, para perceber objetos e o meio, e **robótica** para que a máquina seja capaz de manipular objetos (RUSSELL; NORVIG, 2009). Estas seis capacidades necessárias para atender o teste total de Turing definem as seis disciplinas que compõem a maior parte da área de inteligência artificial.

Diferentemente da linha de ação humana, onde o foco está em emular capacidades humanas, a proposta de ação racional segue a linha de **agentes inteligentes** ou racionais, programas computacionais que têm a capacidade de operar de forma autônoma, perceber o seu ambiente, persistir conhecimento por um longo período de tempo, se adaptar à mudança, criar e então buscar objetivos (RUSSELL; NORVIG, 2009).

O uso de agentes inteligentes tende a ser favorecido sobre as escolas de IA que defendem o "pensamento humano" porque, para os objetivos de um agente, é necessário somente que este seja capaz de inferir informação corretamente e execute ações de forma racional, o que é mais alinhado com o desenvolvimento científico tradicional do que o conceito de alcançar o pensamento ou comportamento humano (RUSSELL; NORVIG, 2009).

Em termos simples, um agente pode ser qualquer entidade que seja capaz de perceber seu ambiente com o uso de **sensores** e atuar sobre ele por **atuadores**. Assim, podem ser considerados como agentes um robô que se movimenta após detectar algo com sua câmera, ou um software que realiza um processamento após detectar o recebimento de um pacote de rede. Um agente racional, no entanto, é capaz de selecionar qual ação deve tomar para maximizar o seu desempenho, utilizando como evidência os dados e a sequência em que eles foram obtidos pelos seus sensores (RUSSELL; NORVIG, 2009).

É considerado que um agente está **aprendendo** caso ele esteja aprimorando sua *performance* em novas tarefas após coletar dados do ambiente. O aprendizado autônomo de aplicações é importante por três principais motivos: Os desenvolvedores não conseguem prever todas as possíveis situações que a aplicação irá se deparar; não é possível prever como o ambiente será modificado com o tempo; para alguns problemas, é difícil para os desenvolvedores implementar a solução do problema em si, sendo mais fácil implementar um raciocínio que leve a própria aplicação a solucionar o problema (RUSSELL; NORVIG, 2009).

A área de Aprendizagem de Máquina é a disciplina de Inteligência Artificial que tem como foco agentes e aplicações que são capazes de aprender, em outras palavras, é o estudo sistemático de sistemas que conseguem aprimorar seu desempenho a partir de experiências (FLACH, 2012). Para os objetivos deste projeto, o foco será nesta disciplina, cujos conceitos são aprofundados nas próximas sessões.

2.2.2 Aprendizagem de Máquina

Com a enorme quantidade de dados disponíveis atualmente, os chamados "*big data*", a área de Aprendizagem de Máquina tem se tornado cada vez mais relevante. Diferentemente de problemas lógicos, onde é possível desenvolver um algoritmo tradicional para solucioná-los, existem problemas onde não é possível definir uma linha de raciocínio lógico. Alguns exemplos de tais problemas são a predição de qual produto é mais interessante para um consumidor, ou a definição de qual e-mail pode ser um *spam*. Para estes problemas, apesar de não existir um algoritmo conhecido, existem quantidades gigantescas de dados (ALPAYDIN, 2014).

A Aprendizagem de Máquina permite que sejam desenvolvidos sistemas capazes de desenvolver seus próprios algoritmos a partir do processamento de um volume muito grande de dados. Com ela é possível que sejam desenvolvidos sistemas de software para otimizar um critério de desempenho a partir de exemplos. Essa maneira peculiar de desenvolver sistemas permite generalizar o aprendizado, a ponto de que seja possível que estes tipos de software sejam capazes de fazer predições sobre o futuro, com base em padrões identificados nos dados das experiências passadas (AWAD; KHANNA, 2015).

O desenvolvimento de aplicações de Aprendizagem de Máquina pode ser abordado de diversas formas, mas em termos gerais estes sistemas de software são compostos por três ingredientes principais: **recursos**, **modelos** e **tarefas** (FLACH, 2012). Recursos são valores ou medidas que podem ser acessados rapidamente pelas aplicações de Aprendizagem de Máquina. Recursos simples frequentemente usados são os conjuntos de números reais e inteiros, que podem ser utilizados para classificar ou contabilizar uma ocorrência detectada pela aplicação (FLACH, 2012).

Modelos formam o principal conceito em Aprendizagem de Máquina, pois eles consistem no conhecimento que é criado pelo processamento de dados. Existem diversos tipos de modelos

e por se tratar de uma área em expansão, estes tipos tendem a aumentar com o tempo. No entanto, é possível agrupá-los em três categorias (FLACH, 2012): **modelos geométricos** são modelos utilizados para conhecimentos que possam ser classificados de forma geométrica, seguindo conceitos de linhas, planos ou distâncias, como, por exemplo um grupo de dados que possa ser apresentado em um sistema de coordenadas cartesiano; **modelos probabilísticos** são usados para conhecimentos estatísticos, por exemplo, dada uma variável X, qual a probabilidade que ela leve até o resultado Y; **modelos lógicos** são usados para conhecimentos mais próximos de algoritmos convencionais, como conjuntos de regras ou árvores de decisão.

Tarefas são os tipos de problemas que a aplicação deve resolver, por exemplo no caso de verificar se um e-mail é ou não um *spam*, isso é considerada uma tarefa de **classificação**. Se o objetivo for definir a probabilidade de que o e-mail seja um *spam* em uma escala de 0 a 10, isso é chamado de uma tarefa de **regressão**, pois demanda que a aplicação aprenda definições reais de valores a partir de exemplos. Existem também tarefas de ***ranking*** que envolvem a ordenação de itens de acordo com algum critério, ***clustering*** na qual itens são particionados em regiões homogêneas e **redução dimensional** onde a representação de um item é convertida para uma de uma dimensão menor (MOHRI; ROSTAMIZADEH; TALWALKAR, 2012).

As tarefas de classificação, regressão e *ranking* fazem parte de um mesmo grupo de problemas, chamados de problemas de **aprendizado supervisionado**. Em termos simples, este é o nome dado para tarefas onde existe uma entrada **X** e uma saída **Y** e o objetivo é aprender o mapeamento entre ambos. Esta forma de aprendizado recebe este nome pois utiliza dados com **rótulos**, ou do inglês *labels*, que são dados cujas classificações são conhecidas e podem ser usados para realizar uma fase de **treinamento**. A partir da base de dados de treinamento, a aplicação é capaz de criar um modelo de predição, possibilitando que ela infira as classificações de uma base de **teste**, cujos rótulos são desconhecidos (AWAD; KHANNA, 2015).

Em contraste ao aprendizado supervisionado, existem os problemas de **aprendizado não supervisionado**, que englobam as tarefas de *clustering* e redução dimensional. A principal característica deste grupo é que ele não utiliza dados rotulados, de forma que as aplicações têm apenas as entradas como recurso de aprendizado. Neste tipo de problema, o objetivo é encontrar padrões e similaridades entre as entradas de forma que seja possível agrupá-los (ALPAYDIN, 2014).

Para se criar grupos a partir de dados de entrada não rotulados, é necessário o uso de algoritmos de *clustering*, ou agrupamento. Estes algoritmos são geralmente iterativos e atuam sobre a presunção de semelhança entre seus dados de entrada. Dentre os algoritmos de *clustering* mais populares estão: ***k-Means***, no qual os dados são divididos em k grupos e ***Expectation-Maximisation***, que permite o agrupamento considerando variáveis ocultas ou inexistentes, a partir do cálculo de uma métrica de expectativa (FLACH, 2012).

A aplicação de *clustering* depende da seleção de uma função que seja capaz de medir o **grau de similaridade** entre pares de dados (AWAD; KHANNA, 2015) e a partir desta métrica

alocá-los em *clusters* ou grupos. Para os objetivos deste estudo, a ênfase será dada na métrica NCD, que é usada pela ferramenta DAMICORE (SANCHES; CARDOSO; DELBEM, 2011), componente essencial da metodologia Tricorder.

A métrica NCD pode ser considerada única por não depender de uma estrutura de dados específica, pois ela é capaz de utilizar diretamente a composição binária destes dados. Tal informação binária, também chamada de ***string*** é inserida em um compressor de arquivos e então uma medida é extraída a partir da forma em que os dados foram comprimidos (SANCHES; CARDOSO; DELBEM, 2011).

Os compressores são capazes de compactar grandes conjuntos de dados em poucas informações, utilizando similaridades entre eles. O NCD consegue extrair uma distância entre os dados a partir da forma em que eles foram comprimidos, sendo que esta métrica suporta uma grande variedade de compressores que utilizam diferentes regras para detectar similaridades.

Os termos apresentados anteriormente fazem parte do jargão da área de Aprendizagem de Máquina, utilizado ao decorrer deste estudo. Para fins informativos esta terminologia é apresentada a seguir, conforme definida por Awad (AWAD; KHANNA, 2015) e Mohri (MOHRI; ROSTAMIZADEH; TALWALKAR, 2012):

- **Amostra de treinamento**: Dados usados para treinar uma aplicação;
- **Amostra de validação**: Dados usados para calibrar os parâmetros de uma aplicação de Aprendizagem de Máquina;
- **Amostra de treino**: Dados usados para avaliar o desempenho de uma aplicação;
- **Matriz de confusão**: Também chamada de matriz de erro, é utilizada para avaliar o desempenho de uma aplicação de classificação, apresentando o número encontrado e desejado de positivos e negativos;
- **Acurácia**: Também chamada de taxa de erro, é a porcentagem de acerto das predições realizadas pela aplicação;
- **Validação cruzada**: Técnica de verificação que avalia o desempenho da aplicação utilizando um conjunto de dados, o dividindo em amostras de treino e teste;
- **Dimensão**: Um grupo de atributos que define uma propriedade. Usado para filtrar, classificar e agrupar dados;
- **Descoberta de conhecimento**: Processo de abstrair conhecimento coleta de fontes estruturadas ou não, para que este conhecimento possa ser usado como base para novos aprendizados;
- **Esquema**: Especificação de alto-nível dos atributos e propriedades de um conjunto de dados;

- **Vetor de recursos**: Vetor de n-dimensões com variáveis que possam ser usadas como recursos para uma aplicação.

2.2.3 Campos de Aprendizagem de Máquina

As áreas de IA e Aprendizagem de Máquina possuem diversos subcampos com focos em diferentes tipos de inteligência e aprendizado. Neste trabalho, três campos recebem destaque por serem utilizados nos experimentos descritos posteriormente. Estes são: **Aprendizagem Profunda** e **Aprendizagem por Reforço** que são subcampos de Aprendizagem de Máquina, e **Processamento de Linguagens Naturais**, um campo de Inteligência Artificial que é frequentemente utilizado em conjunto com Aprendizagem de Máquina (DEVLIN *et al.*, 2019).

Segundo Kirk (KIRK, 2017), o subcampo de Aprendizagem Profunda, do inglês *Deep Learning*, é uma evolução do conceito computacional de redes neurais, representações matemáticas do cérebro humano, propostas em 1943 por McCulloch e Pits (MCCULLOCH; PITTS, 1943). Este estudo introduz o conceito de lógica de limiares, que diz que um neurônio matemático altera sua saída quando a combinação de suas entradas atinge um limiar, assim classificando estas entradas. A partir destes trabalhos foi proposto o termo Rede Neural Artificial, do inglês *Artificial Neural Network* (ANN) que considera diversas camadas de neurônios, interconectados entre si, de forma que a saída de um neurônio pode ser usada como entrada de múltiplos outros (RUSSELL; NORVIG, 2009).

Após sua introdução, o conceito de Redes Neurais Artificiais foi objeto de outros estudos que promoveram sua evolução, tendo como objetivo corrigir ou contornar limitações da proposta original. Finalmente, o termo Aprendizagem Profunda foi alcançado pois, a partir destes esforços, a recente disponibilidade de maior poder computacional e a criação de algoritmos mais eficientes de treinamento, foi proposta uma variação das Redes Neurais Artificiais, chamada de Rede Neural Profunda, cujo nome se refere ao uso de diversas camadas de neurônios para realizar o aprendizado (AWAD; KHANNA, 2015).

Aprendizagem por Reforço segue uma filosofia diferente dos demais subcampos de Aprendizagem de Máquina. Considerando um jogo de xadrez, por exemplo, um agente de aprendizado supervisionado precisa receber entradas que já contenham a informação se um movimento é correto ou não, porém no contexto de xadrez, essa não é uma informação disponível. Na ausência destas informações nas entradas, o agente necessita encontrar outra forma de identificar se um movimento é bom ou ruim, de forma que possa aprender. No contexto de xadrez, essa informação pode ser retirada do fato do agente ganhar ou perder o jogo, o que é descoberto dinamicamente. Este é o conceito chave de Aprendizagem por Reforço, onde o agente recebe um retorno positivo ou negativo de acordo com suas ações e aprende com eles (RUSSELL; NORVIG, 2009).

Na Aprendizagem por Reforço, os agentes têm como objetivo maximizar os retornos

positivos, também chamados de recompensas, a fim de encontrar as melhores ações a serem tomadas. Neste subcampo, o modelo de aprendizado utiliza as recompensas anteriores para decidir quanto a explorar novas possibilidades, ou utilizar as ações anteriores, que já trouxeram resultados positivos, buscando encontrar os melhores resultados e explorar novas possibilidades (MOHRI; ROSTAMIZADEH; TALWALKAR, 2012).

Por fim, o campo de Processamento de Linguagens Naturais é um dos principais campos de Inteligência Artificial, descrito anteriormente como uma das habilidades necessárias para o teste de Turing. Linguagens naturais, como português ou inglês, não podem ser caracterizadas como um conjunto definitivo de sentenças, como acontece com linguagens formais, como linguagens de programação, por exemplo (RUSSELL; NORVIG, 2009).

Linguagens naturais possuem vários fatores que dificultam seu entendimento, como o fato de serem ambíguas, estarem constantemente mudando e por possuírem uma grande quantidade de palavras, combinações e variações. Porém, mesmo com estas dificuldades, para que um agente seja capaz de realizar a aquisição de conhecimento a partir de linguagens humanas, é necessário que ele seja capaz de realizar o Processamento de Linguagens Naturais. Este processamento pode ser realizado com diversos objetivos, como a classificação de textos, por exemplo, para verificar se uma mensagem é um *spam*, ou coleta de informações, para por exemplo, encontrar as informações que sejam mais relevantes para os interesses de um usuário (RUSSELL; NORVIG, 2009).

Devido à grande dificuldade e complexidade de se realizar o processamento de linguagens naturais, o campo de Aprendizagem de Máquina tem sido usado recorrentemente em projetos de Processamento de Linguagens Naturais. Isto pode ser justificado, pois na literatura podem ser encontradas várias aplicações de Aprendizagem de Máquina com sucesso para a resolução de problemas complexos, sem que seja necessária a criação de algoritmos e programas específicos, tornando este campo um ótimo candidato para ser utilizado em conjunto com o Processamento de Linguagens Naturais (MARIE-SAINTE *et al.*, 2019).

2.2.4 Problemas em Aprendizagem de Máquina

As aplicações de Aprendizagem de Máquina possibilitam a solução de diversos tipos de problemas, porém a flexibilidade dessa área gera dificuldades, como por exemplo, garantir que o problema correto está sendo solucionado ou mesmo que qualquer problema está sendo solucionado. Na literatura ainda são recentes os recursos para o desenvolvimento de código deste tipo de software, como padrões, boas práticas e testes (LAKSHMANAN; ROBINSON; MUNN, 2020), sendo que até recentemente estes recursos eram limitados e difíceis de encontrar (KIRK, 2014). Por estes motivos, a garantia de qualidade nesta área pode ser considerada como um desafio.

Teoricamente, algoritmos de Aprendizagem de Máquina podem ser considerados robustos

por serem capazes de aprender com erros passados e evitar novos problemas ao decorrer do tempo. Infelizmente isso não é necessariamente correto para humanos. Apesar destas aplicações serem capazes de minimizar erros, elas ainda dependem de que seus desenvolvedores tenham utilizado os dados corretos e não tenham cometido enganos em seus códigos. A maior dificuldade nesta área de IA é encontrar técnicas de testes que possam auxiliar na detecção destes tipos de problemas (KIRK, 2014).

Segundo Kirk (KIRK, 2014), devido à importância do uso de dados nesta área, as aplicações de Aprendizagem de Máquina estão sujeitas a um grupo específico de problemas relacionados à qualidade da base de dados utilizada:

- **Dados instáveis**: Os algoritmos são capazes de evitar dados instáveis em uma base de dados, porém caso os dados fornecidos pelos desenvolvedores tenham problemas de formatação ou estiverem incorretos, isso pode causar um aprendizado errado;
- ***Underfitting***: Este é o nome dado quando um modelo não consegue capturar a tendência de um conjunto de dados. Isto pode ocorrer pois o modelo não tem dados suficientes para alcançar a acurácia desejada ou quando os dados não têm uma qualidade boa, de forma que a eficiência do algoritmo seja prejudicada, apresentando baixa variação e alta tendência;
- ***Overfitting***: Este é o nome dado quando o modelo passa a considerar ruídos e dados incorretos da base de dados. Isto pode ocorrer quando o modelo tenta abranger todos os dados ou recebe uma quantidade excessiva de dados, fazendo com que o modelo não classifique dados corretamente, tendo uma alta variação e baixa tendência;
- **Futuro imprevisível**: Caso a base de dados não cubra um leque suficiente de possibilidades, a classificação de um novo dado, que seja completamente inédito para o modelo, tende a ser difícil para o algoritmo e até mesmo para a avaliação de um analista.

Os problemas listados podem ser causados por uma base de dados mal selecionada, mas é possível também que a base seja adequada e os problemas estejam sendo causados por defeitos no código, fazendo com que os dados inseridos sejam uma versão alterada da base.

A busca por defeitos em uma aplicação de Aprendizagem de Máquina é uma tarefa difícil, pois assim como em códigos legados, em muitos casos não é simples entender o que está acontecendo internamente na aplicação (KIRK, 2014). Técnicas tradicionais de teste como testes estruturais não têm uma grande eficiência neste contexto, pois é comum o uso de bibliotecas ou ferramentas especializadas, dificultando ou até mesmo impossibilitando a verificação interna do código de Aprendizagem de Máquina. Uma abordagem alternativa são os testes funcionais, os quais tratam as aplicações como caixas-pretas.

Mesmo os testes funcionais convencionais encontram dificuldades em IA, pois estas aplicações são caracteristicamente difíceis de se validar com base em suas entradas e saídas.

Como elas são capazes de criar seus próprios modelos de aprendizado e seus algoritmos, a saída correta pode não ser conhecida pelo testador, tornando a atividade de teste ainda mais onerosa. Assim, é necessário buscar formas alternativas e complementares para detectar possíveis problemas no software.

Devido às peculiaridades de Aprendizagem de Máquina, existem técnicas de teste mais recomendadas, como a validação cruzada, descrita anteriormente, e o monitoramento de parâmetros. O monitoramento do tempo em que a aplicação executa a etapa de treinamento, por exemplo, é uma forma de verificar se estão ocorrendo *underfitting* ou *overfitting*, pois se em uma execução da aplicação ela apresentar um tempo de treinamento significativamente menor ou maior, isso pode ser um indício de um desses defeitos (KIRK, 2014).

Nesse sentido, observa-se que o comportamento da execução da aplicação pode ser usado como um parâmetro valioso na detecção de defeitos do código. Este estudo explora esta hipótese, analisando o comportamento da execução da aplicação, no que tange ao uso pela mesma de recursos computacionais.

2.3 Validação, Verificação e Teste de Software

O processo de desenvolvimento de software possui uma grande variedade de procedimentos, ferramentas, boas práticas e padrões que promovem a qualidade e evitam falhas em um software. Porém, mesmo com a utilização destes recursos, a atividade de desenvolvimento tende a se tornar complexa, exigindo um grande investimento de horas e volume de código, tornando propensa a existência de problemas que podem resultar em um produto final diferente do esperado (DELAMARO; JINO; MALDONADO, 2016).

Dada a complexidade da atividade de desenvolvimento de software, problemas podem ocorrer durante todas suas fases, desde a definição de requisitos até a real implementação. Considerando isto, a realização de testes é extremamente importante para que seja possível identificar e eliminar estes erros (SOUZA *et al.*, 2000).

Nesta seção são apresentadas definições de validação, verificação e teste e como estas estão vinculadas a este projeto.

2.3.1 Definições Básicas

A atividade de teste de software é composta por um conjunto de processos que tem como objetivo verificar e validar se um programa, ou parte do código de um programa, realiza o que foi especificado para fazer e nada diferente disso (MYERS; SANDLER; BADGETT, 2011). Estes processos de teste são chamados coletivamente de VV&T (DELAMARO; JINO; MALDONADO, 2016).

O real significado dos termos validação e verificação é frequentemente invertido e em

alguns casos, ambos são até mesmo tratados como equivalentes, porém estes dois termos não se referem às mesmas atividades (SOMMERVILLE, 2010). Barry Boehm (BOEHM, 1979) apresenta uma definição clara e sucinta para ambos:

```
Validação: Se o produto correto está sendo desenvolvido.
Verificação: Se o produto está sendo desenvolvido corretamente.
```

De acordo com suas definições, as atividades de VV&T devem ser realizadas em todas as etapas do processo de desenvolvimento de software, não somente no produto final. Estas atividades podem ser divididas em duas categorias principais: estáticas e dinâmicas (DELAMARO; JINO; MALDONADO, 2016).

As atividades de VV&T chamadas dinâmicas são as que envolvem a execução de um programa ou modelo, enquanto as atividades estáticas não necessitam da existência de executáveis para serem realizadas (DELAMARO; JINO; MALDONADO, 2016). Neste projeto são utilizadas exclusivamente atividades dinâmicas que utilizam diferentes tipos de entrada e então verificam o comportamento da aplicação sendo testada.

No contexto de software, existe uma distinção importante entre alguns termos que são utilizados, frequentemente de forma errada, para se referir a problemas encontrados em um projeto ou produto. Estes termos são: defeito, engano, erro e falha, e apesar de muitas vezes serem tratados como sinônimos, eles possuem significados importantes e significativamente distintos.

Segundo o padrão internacional publicado pela IEEE (ISO/IEC/IEEE..., 2017) de vocabulário de engenharia de software e sistemas, estes termos têm os seguintes significados:

- **Engano** (do inglês, *mistake*): Ação humana que insere um defeito no código;
- **Defeito** (do inglês, *fault*): é um passo, processo ou definição de dados incorreto, incompleto, ausente ou extra que, ao ser executado, pode produzir um erro no programa;
- **Erro** (do inglês, *error*): ocorre quando um defeito é executado e um estado incorreto é atingido durante a execução do programa. Em outras palavras, um erro é qualquer estado intermediário ou final na execução do programa que seja inconsistente com o esperado;
- **Falha** (do inglês, *failure*): evento notável em que o sistema viola a sua especificação, ou seja, produção de uma saída incorreta em relação à especificação.

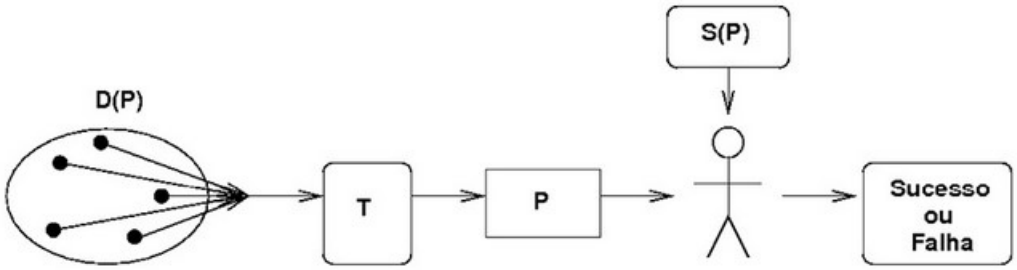
Além destes termos, existem outras definições que são utilizadas comumente no contexto de teste de software. Estas terminologias, apesar de não serem listadas explicitamente pela IEEE como as apresentadas anteriormente, são de suma importância na área de VV&T e são utilizadas neste documento, assim é interessante a apresentação destes outros termos e seus significados, conforme definidos por Delamaro et al. (DELAMARO; JINO; MALDONADO, 2016):

- **Domínio de entrada**: Conjunto de todos os valores que possam ser utilizadas como entrada para um programa P. Denotado por D(P);
- **Domínio de saída**: Conjunto de todos os valores que possam ser resultados da execução de um programa P;
- **Dado de teste**: Um elemento que faz parte do domínio de entrada de P;
- **Caso de teste**: Par de informações, formado pelo dado de teste e seu respectivo resultado esperado, conforme a especificação do programa (S(P)). Denotado por T;
- **Conjunto de teste** ou conjunto de casos de teste: Conjunto de todos casos de teste que foram realmente utilizados em uma atividade de teste.

Os termos listados anteriormente permitem o desenho de um fluxo de atividade de teste, faltando apenas um papel crucial, o testador, que tem como responsabilidade analisar os resultados do programa, cruzá-los com a especificação e então indicar se o resultado está correto ou não. Este testador é chamado de **oráculo de teste**.

Um exemplo de fluxo de atividade de teste é apresentado na Figura 2, utilizando as siglas da terminologia. No fluxo ilustrado, a partir de um domínio de entrada D(P) é criado um caso de teste T, que é utilizado para a execução de um programa P. O oráculo verifica as saídas do programa e as compara com a especificação do programa S(P). Caso as saídas não coincidam com o esperado conforme a especificação, então um erro é identificado (DELAMARO; JINO; MALDONADO, 2016).

Figura 2 – Cenário típico da atividade de teste.



Fonte: Delamaro, Jino e Maldonado (2016).

O cenário apresentado na Figura 2 demonstra uma atividade de teste que é realizada dentro de uma **sessão de teste** ou sessão de trabalho. Em uma sessão de teste diferentes execuções podem ser feitas com diferentes casos de teste. Nesse sentido, ela é usada para definir as atividades de teste, as quais podem ser quebradas em etapas, armazenando estados intermediários para que possam ser verificados posteriormente pelo testador (SOUZA *et al.*, 2000).

2.3.2 Tipos de Teste

Dada a abrangência e complexidade da atividade de teste de software, é necessário que esta seja dividida em fases com diferentes granularidades. Estas fases foram criadas com o objetivo de cobrir diferentes tipos de problemas que podem ser encontrados durante o processo de desenvolvimento de software. Assim, a primeira fase tem como objetivo encontrar erros localizados nas menores unidades de um programa, as funções ou rotinas do código, a segunda busca problemas que possam ocorrer quando diferentes partes do programa passam a interagir umas com as outras, e a terceira tem como foco encontrar falhas no atendimento de requisitos após a integração de todas as partes de código. Estas três fases são chamadas, respectivamente, de **teste de unidade**, **teste de integração** e **teste de sistemas** (DELAMARO; JINO; MALDONADO, 2016).

Teste de unidade é o nome dado ao processo de testar os menores componentes de um programa, como métodos ou classes de objetivo, no caso de linguagens orientadas a objeto (SOMMERVILLE, 2010). Estes testes buscam problemas relacionados a algoritmos ou estruturas de dados errados ou erros de implementação (DELAMARO; JINO; MALDONADO, 2016). A fase de teste unitário é a primeira a ser executada, pois permite encontrar problemas isolados em unidades de código, antes que estes possam causar problemas nas integrações, e pode ser executada rapidamente, logo após a implementação de uma unidade de código (MYERS; SANDLER; BADGETT, 2011).

Teste de integração é a fase realizada após os testes unitários, pois verificam a integração entre as unidades de código. Essa fase foca em testar a interface entre componentes do software (SOMMERVILLE, 2010). Conforme diferentes partes de software são integradas e passam a fazer interface, é necessário verificar se essa interação está funcionando conforme planejado, sem a presença de erros. Assim como a fase de testes de unidade, é necessário conhecimento do código interno e neste caso específico, também é necessário o conhecimento da estrutura do software. Por este motivo, ambas as fases de testes são normalmente executadas pelos próprios desenvolvedores (DELAMARO; JINO; MALDONADO, 2016).

A fase de **teste de sistema** é realizada após a integração de todas as partes do software, de forma que este possa ser considerado como completo. Ela tem como objetivo verificar se todas as funcionalidades definidas no documento de requisitos estão implementadas e funcionando conforme especificado (DELAMARO; JINO; MALDONADO, 2016). É impossível realizar testes de sistema sem que haja um documento onde todos os objetivos mensuráveis dos produtos estão escritos (MYERS; SANDLER; BADGETT, 2011).

Testes de sistema podem contemplar diversas categorias, de acordo com as necessidades que foram especificadas no documento de requisitos. Myers, Sandler e Badgett (MYERS; SANDLER; BADGETT, 2011) listam quinze categorias diferentes de casos de teste, das quais, as mais frequentemente encontradas são usabilidade, segurança, desempenho e confiabilidade.

- **Usabilidade**: Testes para determinar o quão bem o usuário final consegue interagir com o programa;
- **Segurança**: Testes para corromper as medidas de segurança de um programa;
- **Desempenho**: Testes para determinar se o programa atende os requisitos de desempenho estabelecidos;
- **Confiabilidade**: Testes para determinar se o programa alcança os requisitos de confiabilidade.

Apesar do foco deste estudo se relacionar fortemente a desempenho, tal foco não tem como objetivo o uso de testes de desempenho; mas sim o uso de desempenho para testes de software, visando encontrar defeitos de software que produzem comportamentos diferentes dos desejados ou especificados na aplicação.

As três fases de testes apresentadas são utilizadas principalmente durante o processo convencional de desenvolvimento, porém após a entrega de um software é iniciada a fase de suporte e manutenção, que tende a ser a fase mais longa no ciclo de vida do desenvolvimento de um software comercial (AMMANN; OFFUTT, 2008). Durante a fase de manutenção são executados os chamados **testes de regressão**, que têm como objetivo verificar se eventuais modificações estão corretas e se os requisitos testados anteriormente continuam sendo atendidos (DELAMARO; JINO; MALDONADO, 2016).

Além de ser extensa, a fase de manutenção de um software é a mais propensa à inserção de novos defeitos (MYERS; SANDLER; BADGETT, 2011). Por este motivo os testes de regressão são cruciais para a garantia de qualidade do software. Apesar disso, estes tendem a ser focados na reexecução de casos de testes usados durante o desenvolvimento, sendo possível que defeitos mais complexos, que podem não afetar diretamente as saídas de uma aplicação, não sejam detectados.

A metodologia Tricorder abordada neste estudo tem uma forte aderência com a fase de manutenção de software, podendo ser utilizada como um complemento aos testes de regressão. Além do processo tradicional de manutenção, o campo chamado MLOps (*Machine Learning Operations*) passou a ganhar destaque nos últimos anos. Ele reúne práticas e procedimentos com o objetivo de possibilitar entregas contínuas de software de Aprendizagem de Máquina (MÄKINEN *et al.*, 2021).

O processo de entrega contínua é frequentemente usado no desenvolvimento de software convencional, com o uso de processos e ferramentas, abrangendo inclusive abordagens de VV&T. MLOps visa trazer os benefícios desta prática, porém com especificidades do campo de Aprendizagem de Máquina (MÄKINEN *et al.*, 2021). Desta forma a metodologia Tricorder pode ser uma candidata a ser incorporada em um processo de MLOps, de forma a verificar automaticamente a existência de anomalias durante o processo de entrega de software.

2.3.3 Técnicas de Teste

Umas das principais etapas da atividade de VV&T e que é capaz de definir se esta vai ser bem-sucedida ou não, é a seleção do conjunto de casos de teste (SOUZA *et al.*, 2000). Na maioria das vezes é infactível realizar testes de forma "exaustiva", ou seja, utilizando todas as possíveis entradas. Por este motivo é vital para a atividade de teste que o conjunto de teste selecionado tenha a qualidade necessária para revelar defeitos desconhecidos no código (DELAMARO; JINO; MALDONADO, 2016).

Existem maneiras sistemáticas para se selecionar conjuntos de testes que possam ser eficientes na identificação de erros, conforme as restrições de custo e tempo de cada projeto. Estas maneiras são apresentadas por **critérios de teste** (SOUZA *et al.*, 2000). Estes critérios podem ser classificados em quatro categorias, também chamadas de **técnicas de teste**, que se diferem pela origem da informação utilizada para avaliar os conjuntos de teste: **Funcional**, **Estrutural**, **Baseado em Erros** e **Baseado em Modelos** (DELAMARO; JINO; MALDONADO, 2016).

A técnica de **teste funcional** tem como objetivo verificar as funcionalidades de um software e não sua implementação, assim, o testador não precisa interagir com o código diretamente, tratando a aplicação como uma caixa-preta. Por este motivo, essa técnica também pode ser chamada de teste de caixa-preta (SOMMERVILLE, 2010).

Os critérios que fazem parte da técnica de teste funcional se baseiam exclusivamente na especificação do software, de forma que a qualidade destes critérios está diretamente ligada à qualidade dos requisitos. Um benefício desta técnica é que ela pode ser aplicada em qualquer fase de testes e qualquer tipo de software, pois nela os detalhes de implementação não são relevantes (DELAMARO; JINO; MALDONADO, 2016).

A execução de um conjunto de testes que contemple todas as possíveis entradas permitidas pelos requisitos do software é inviável, devido ao seu custo de tempo e computacional. Idealmente deve-se definir um subconjunto reduzido do domínio de entrada que contenha dados de teste de todos os subdomínios que sejam semelhantes, de forma que todos os "tipos" de entrada possam ser representados (DELAMARO; JINO; MALDONADO, 2016).

Para se realizar a criação de subconjuntos de entrada foram desenvolvidos critérios de teste, que propõem abordagens sistemáticas para definir quais entradas devem ser selecionadas. Segundo Delamaro et al (DELAMARO; JINO; MALDONADO, 2016), os critérios de teste funcional mais conhecidos são:

- **Particionamento em classes de equivalência**: Propõe a divisão do domínio de entrada do sistema sob teste, criando conjuntos que possam ser considerados equivalentes conforme a especificação do programa. Desta forma, é possível a realização abrangente de testes, utilizando apenas uma entrada de cada conjunto criado;

- **Análise de valor limite**: Usado em conjunto com o particionamento de equivalência, propõe a utilização de entradas que estejam nos limites dos conjuntos criados pelo particionamento. Pois acredita-se que casos de testes com valores limites têm uma eficiência maior na detecção de defeitos;

- **Grafo causa-efeito**: Utiliza um grafo, criado a partir das especificações do software, onde as entradas correspondem às causas e as saídas aos efeitos. Diferentemente dos critérios anteriores, é capaz de cobrir combinações de entradas.

Diferentemente do teste funcional, o **teste estrutural** necessita que haja conhecimento sobre o código do programa para que a sua estrutura possa ser usada na definição dos testes (SOMMERVILLE, 2010). Esta técnica, também chamada de caixa-branca, define casos de teste que executam partes ou componentes elementares do programa (MESKINI; NASSIF; CAPRETZ, 2013).

A técnica estrutural busca testar os caminhos lógicos de um software, criando casos de teste que verificam conjuntos de condições, laços, definições e até mesmo o uso de variáveis (DELAMARO; JINO; MALDONADO, 2016).

Por necessitar de conhecimento sobre o código-fonte, o teste estrutural sofre algumas limitações. Não é possível, por exemplo, a criação de um procedimento de teste com propósito geral, pois o código de cada software contém suas próprias e específicas peculiaridades, isto aumenta a dificuldade de se automatizar este tipo de técnica (DELAMARO; JINO; MALDONADO, 2016).

Assim como no teste funcional, foram desenvolvidos critérios com abordagens sistemáticas para a realização de testes estruturais. De maneira geral, os critérios de teste estrutural utilizam **Grafos de Fluxo de Controle** (GFC), que são representações gráficas do programa onde seus nós são blocos de código e as arestas representam o seu fluxo de controle. Estes critérios podem ser divididos em três grupos (DELAMARO; JINO; MALDONADO, 2016):

- **Baseados na complexidade**: Propõem requisitos de testes com base nos dados de complexidade do sistema sob teste. O critério mais conhecido deste grupo é o teste de caminho básico, também chamado de McCabe, que utiliza a complexidade ciclomática do GFC, a métrica de complexidade lógica do software;

- **Baseados em fluxo de controle**: Utilizam informações do controle de execução do programa para definir seus requisitos de teste. Os critérios mais populares deste grupo são:

  - **Todos-Nós**: A execução do teste deve passar por todos os nós do GFC da aplicação;
  - **Todos-Arestas**: Todas as arestas do GFC devem ser executadas ao menos uma vez durante a execução do teste;

  - **Todos-Caminhos**: Todos os caminhos possíveis devem ser percorridos na execução do teste.

- **Baseados em fluxo de dados**: Propõem requisitos de teste a partir do fluxo de dados da aplicação, que contempla a definição e o eventual uso de variáveis ao decorrer da aplicação. Os critérios mais usados deste grupo são:

  - **Todas-Definições**: Todas as variáveis devem ser definidas ao menos uma vez durante o teste;
  - **Todos-Usos**: Todas as variáveis definidas devem ser utilizadas ao menos uma vez;
  - **Todos-Du-Caminhos**: Todas associações entre a definição e o uso de uma variável devem ser percorridas por todos os caminhos onde esta variável não é redefinida.

Os **testes baseados em defeitos** contemplam diferentes tipos de teste, dentre os quais se destacam **teste de mutação** e **injeção de erros** (DELAMARO; JINO; MALDONADO, 2016).

O teste de mutação ou análise de mutantes é um critério de teste que altera o programa sendo testado, de forma que um novo conjunto de programas alternativos, também chamados de mutantes, sejam criados contendo os defeitos inseridos no código original. Com esta técnica de teste é possível verificar subdomínios de entrada que possam distinguir o programa original de um mutante, e, além disso, um conjunto de casos de teste que seja capaz desta distinção tende a ser capaz de revelar outros tipos de defeito (DELAMARO; JINO; MALDONADO, 2016).

O critério de injeção de erros ou semeadura de erros tem como objetivo encontrar a razão de erros originais e erros inseridos em um código, com o intuito de que esta razão possa representar o número de erros originais ainda presentes no software. Para se encontrar esta razão, primeiro são inseridos defeitos artificiais no programa, sendo que a qualidade dos defeitos inseridos impacta decisivamente na eficácia deste critério. Diferentemente da análise de mutantes onde os defeitos são inseridos de forma sistemática, nesta abordagem os erros tendem a ser inseridos manualmente (SOUZA *et al.*, 2000).

Teste baseado em modelo utiliza o conceito de modelagem, o qual utiliza o conhecimento de um sistema para criar um modelo que captura este conhecimento e permite que este seja reutilizado ao decorrer do desenvolvimento. Com um modelo mais sofisticado, é possível até mesmo utilizá-lo como oráculo de teste (DELAMARO; JINO; MALDONADO, 2016).

Os modelos que podem ser utilizados para teste devem contemplar, idealmente, todas as sequências de ações que ocorrem ao se utilizar o sistema, pois um modelo limitado pode dificultar a automação de testes. Existem técnicas que podem auxiliar na criação de um modelo ideal para testes. Estas técnicas permitem a decomposição de comportamentos complexos e até mesmo o uso de condições e variáveis, de forma que o modelo consiga representar diferentes configurações do programa (DELAMARO; JINO; MALDONADO, 2016).

2.3.4 Execução de Testes

A ação prática de teste de software pode ser definida como um processo formal executado por um time de especialistas onde uma única unidade de software, um grupo de unidades integradas ou um pacote completo de software, são avaliados a partir da sua execução em um computador, seguindo procedimentos aprovados de teste e utilizando casos aprovados de teste (GALIN, 2004).

Apesar da formalidade que deve ser seguida no processo de teste, a atividade de teste está sujeita a problemas quando executada manualmente, como a possibilidade de erro humano e a impossibilidade de reuso. Além disso, a execução de testes de software tende a ser cara e demandar um grande volume de trabalho, com custos que podem chegar a até 50% do custo total do desenvolvimento de software. Esta porcentagem pode ser maior para sistemas críticos (AMMANN; OFFUTT, 2008).

Uma alternativa à execução manual de teste de software, e que pode reduzir significativamente os custos e minimizar a possibilidade de erro humano, é a automação desta atividade. Testes automáticos necessitam de um grande investimento de tempo e esforço humano para serem desenvolvidos, porém trazem uma economia significativa a cada repetição do teste, o que os tornam atrativos especialmente em grandes projetos de software onde testes devem ser executados muitas vezes (CERVANTES, 2009).

Uma das principais formas de automação de testes é por meio de ferramentas criadas especificamente para este fim. Estas ferramentas são projetadas para executar diferentes técnicas de teste (SNEHA; MALLE, 2017), como a Selenium que tem como foco a automação de testes em aplicações web, permitindo a definição de rotinas de iteração com um site e então a verificação dos valores de saída, e o JUnit que permite a realização automática de testes de unidade em aplicações Java.

Apesar da importância de ferramentas de testes para reduzir os custos e aumentar a eficiência do processo de teste, existe ainda um número limitado de ferramentas de teste que possam ser utilizadas em múltiplos domínios ou em tipos de aplicações menos populares, como software embarcado (MUSTAFA; AL-QUTAISH; MUHAIRAT, 2009). Para estas situações é interessante a criação de novas ferramentas que possam ser aplicadas em software de maneira genérica.

Uma possível abordagem para a criação de novas ferramentas é a partir da modelagem de rotinas de testes realizadas manualmente por profissionais de qualidade. Em indústrias específicas, as atividades executadas por estes profissionais tendem a ser exclusivamente manuais, devido à complexidade e o investimento necessário para automatizá-las. A metodologia apresentada neste projeto propõe a automação da atividade de teste realizada por profissionais de qualidade destas indústrias, onde os dados de desempenho de aplicações são monitorados a fim de encontrar comportamentos inesperados e que possam indicar erros no software sendo monitorado. Esta

atividade pode ser feita com certa facilidade por uma equipe treinada, porém não é facilmente modelada, demandando um investimento grande de horas e dinheiro para que ela continue sendo realizada manualmente.

2.4 Monitoração de Sistemas Computacionais

Monitorar e analisar dados de desempenho são atividades essenciais no desenvolvimento de aplicações, seja para atender requisitos de desempenho ou para validar o seu funcionamento. O desempenho é um dos principais critérios a ser avaliado no desenvolvimento de software, onde o objetivo é usualmente alcançar a melhor *performance* com o menor custo (JAIN, 1991).

A monitoração de aplicações é uma das técnicas de aferição, usadas na avaliação de desempenho de sistemas computacionais. Com ela é possível extrair dados de um software, como o seu uso de recursos e seu desempenho. Nesta seção são apresentados conceitos da área de avaliação de desempenho e como estes estão relacionados a este projeto.

2.4.1 Avaliação de Desempenho

A avaliação de desempenho pode ser realizada por três técnicas: **modelagem analítica**, **simulação** e **aferição** (JAIN, 1991). Modelos analíticos e simulações podem ser considerados como parte do grupo de técnicas de modelagem (LUCAS, 1971), porém ambos têm suas especificidades e serão tratados separadamente neste estudo. A seleção de uma técnica de avaliação deve considerar diversos fatores, como tempo disponível, acurácia desejada e até mesmo a fase do projeto em que é aplicada. A aferição, por exemplo, depende da existência de uma versão do sistema que seja utilizável, caso isso não exista, a modelagem analítica e simulação são indicadas (JAIN, 1991).

Modelagem analítica é a técnica de criar um ou mais modelos matemáticos do sistema cujo desempenho se deseja avaliar. A eficiência desta técnica depende do modelo matemático utilizado, sendo que as formas mais frequentemente usadas são com teoria de filas, redes de Petri ou redes de Markov (LUCAS, 1971). A modelagem não depende de um sistema já existente, porém a criação de um modelo sem o uso de dados da aplicação real pode reduzir a sua confiabilidade.

A criação de modelos analíticos tende a ser favorecida pois exige o menor tempo dentre as três técnicas. Esta abordagem permite que sejam realizadas adaptações e ajustes de forma rápida, permitindo uma maior flexibilidade nos experimentos, porém ela usualmente necessita de simplificações e suposições sobre o sistema real, o que limita a acurácia desta abordagem. Modelagem é muito utilizada pois é a técnica mais barata, podendo ser feita matematicamente necessitando principalmente do tempo de um analista (JAIN, 1991).

A técnica de **simulação** tem o maior potencial e flexibilidade para se realizar uma

avaliação de desempenho, mas pode exigir um grande investimento computacional e de tempo caso a simulação seja detalhada (LUCAS, 1971). A simulação é criada com base nos eventos e condições do sistema real, seguindo as etapas de processamento que devem ser executadas. A simulação permite a inclusão de mais detalhes da implementação do sistema real, promovendo uma maior precisão nos seus resultados. Assim como a modelagem analítica, esta técnica pode ser aplicada sem a existência de dados reais de monitoramento, porém, o uso destes dados na construção de uma simulação enriquece significativamente a qualidade e os resultados desta técnica (JAIN, 1991).

A **aferição** é realizada medindo os dados de uma aplicação real. Ela pode ser realizada de três formas: por **prototipação**, que utiliza uma versão simplificada do sistema, mas que preserva suas funcionalidades; por ***benchmarks***, ferramentas que testam o desempenho da aplicação com fins de comparação; ou por **monitoração**, onde a análise é feita a partir de dados reais do sistema, que deve estar finalizado ou ao menos estar próximo de sua conclusão (LUCAS, 1971). A aferição leva mais tempo que a modelagem analítica e menos que a simulação, porém esta técnica depende de um sistema real e está sujeita às complexidades e peculiaridades do sistema sendo monitorado, desta forma o investimento de tempo necessário é o mais volátil dentre as três técnicas (JAIN, 1991). Apesar de os dados serem coletados diretamente da aplicação real, ainda existe o risco de que a acurácia desta técnica seja baixa. Isso ocorre pois esta técnica depende de diversos parâmetros do ambiente, como a definição de cargas de trabalho, tempo de medição e configurações do sistema sendo analisado. A aferição é a técnica mais cara, pois requer equipamentos reais, instrumentação e tempo, mas tem o maior benefício, sendo mais fácil confiar em dados provenientes do sistema real.

As técnicas de avaliação de desempenho não precisam, necessariamente, ser utilizadas com o mesmo objetivo. É possível utilizar as técnicas com fins diferentes, com o objetivo de tirar o maior proveito das vantagens de cada uma. Por exemplo, a modelagem analítica pode ser utilizada para identificar os efeitos causados por cada parâmetro suportado pela aplicação, e a simulação pode ser realizada para buscar parâmetros dentro do domínio de entrada para encontrar combinações ótimas. Além disso, a utilização de uma técnica não impede o uso de outra, assim, podem ser usadas duas ou mais técnicas simultaneamente dados os objetivos das avaliações (JAIN, 1991).

2.4.2 Monitoração de Software

A monitoração é uma das formas de aplicar a técnica de aferição. Ela é aplicada com o uso de monitores de software ou hardware, que são ferramentas utilizadas para verificar atividades ou dados em um sistema. Na maioria dos casos, monitores são utilizados para detectar como um sistema utiliza recursos de desempenho, gerar estatísticas destes dados, analisá-los e fornecer os resultados para o analista de forma organizada (JAIN, 1991). Algumas ferramentas de monitoração são capazes também de identificar dados que possam representar problemas na

aplicação sendo analisada.

De acordo com Jain (JAIN, 1991), a técnica de monitoração utiliza uma terminologia específica. Esta terminologia é utilizada ao decorrer deste trabalho, assim, é interessante apresentá-la:

- **Evento**: Mudança de estado do sistema que é considerada significativa para o monitor;
- ***Trace***: *Log* de eventos, com detalhes que sejam relevantes para o analista, como tempo e tipo de evento;
- ***Overhead***: Nome dado a qualquer perturbação no sistema causada pelo monitor. Essa perturbação pode acontecer como um acréscimo no consumo de um dos recursos de desempenho do sistema, por exemplo. A minimização de *overhead* é altamente desejada em monitores;
- **Domínio**: Conjunto de atividades ou dados que são medidas por um monitor. Métricas de desempenho, como uso de processador (CPU) e memória principal (RAM), são exemplos de dados frequentemente encontrados no domínio de monitores;
- **Taxa de entrada**: Frequência máxima de eventos que o monitor consegue observar;
- **Resolução**: Precisão em que os dados podem ser monitorados, como, por exemplo, a escala de memória que o monitor é capaz de verificar.

Para se realizar o monitoramento, o sistema sob análise deve receber cargas de trabalho de forma controlada. Por este motivo é importante considerar que os resultados da monitoração estão ligados diretamente com as cargas de entrada utilizadas no sistema e, portanto, devem ser considerados alguns pontos antes de realizar as medições (JAIN, 1991):

- Quais são os tipos de carga de trabalho;
- Quais cargas de trabalho são usadas com mais frequência;
- Como cargas de trabalho apropriadas são selecionadas;
- Como as medições de cargas de trabalho podem ser interpretadas;
- Como o desempenho do sistema pode ser monitorado;
- Como utilizar as cargas de trabalho no sistema de forma controlada;
- Como os resultados da avaliação de desempenho devem ser apresentados.

2.4.3 Ferramentas de Monitoração

O uso de ferramentas é essencial na técnica de monitoramento. De acordo com as necessidades do experimento, devem ser escolhidas ferramentas de um de dois tipos: orientada a eventos, onde o armazenamento de métricas acontece sempre que um evento predefinido ocorre, ou por amostragem, onde os dados são salvos em intervalos fixos de tempo, independentemente da ocorrência de eventos (JAIN, 1991).

A monitoração pode ser realizada com medidores de hardware ou de software. Os monitores de hardware são equipamentos reais conectados ao sistema medido, de forma que seja possível verificar com precisão os estados e dados do sistema e com altas taxas de amostragem, sem que haja um *overhead* no sistema medido. Medidores de software utilizam comandos adicionados ao sistema sendo avaliado para que seja possível coletar dados da aplicação ou do sistema operacional.

Para os objetivos deste projeto, são utilizados exclusivamente monitores por software. Isso gera algumas complicações pois estes tendem a ter taxa de entrada menor, resoluções menores e um *overhead* maior. Além disso, são utilizadas somente ferramentas de monitoração por amostragem, coletando o uso de recursos de desempenho de uma aplicação de acordo com uma frequência definida pelo analista. Por estes motivos é importante encontrar ferramentas com baixos *overheads* e que permitam a monitoração de aplicações de forma caixa-preta, sem que seja necessário instrumentar o código ou ter qualquer acesso a ele (JAIN, 1991).

Existe uma ampla variedade de monitores de software disponíveis para uso comercial ou de forma gratuita. Dentre estes se destacam a biblioteca PSutil (RODOLA, 2021) para a linguagem Python, as ferramentas Ganglia (BERKELEY, 2021) e Zabbix (VLADISHEV, 2021). Existem também recursos de monitoramento fornecidos pelos próprios sistemas operacionais, como é o caso do Windows Performance Monitor (MICROSOFT, 2021a).

A biblioteca PSutil permite a monitoração de aplicações e sistemas com o uso de *scripts* Python, facilitando o seu uso em sistemas de software que utilizam esta linguagem. Por estar atrelada ao Python, pode ser usada em múltiplos sistemas operacionais, apresentando diferentes capacidades em cada um. Tem como limitação as métricas que podem ser monitoradas em alguns sistemas operacionais, como no Windows, onde somente é possível coletar métricas convencionais de aplicações, como porcentagem de CPU, acessos de Entrada/Saída (E/S, ou I/O em inglês) e uso de RAM.

Ganglia é uma ferramenta de código aberto amplamente utilizada para a monitoração de sistemas comerciais (BERKELEY, 2021). Desenvolvida para permitir o monitoramento de sistemas distribuídos de alto desempenho, ela é principalmente utilizada em *clusters* e *grids* que utilizam sistemas operacionais Linux. Por se tratar de uma ferramenta mais sofisticada, ela apresenta uma complexidade grande e necessita de um investimento de tempo para preparação do ambiente.

A Zabbix é uma ferramenta de código aberto, porém focada em uso comercial, oferecendo uma grande variedade de recursos de monitoramento, como de aplicações, bases de dados, redes, servidores e máquinas virtuais. Assim como o Ganglia, esta ferramenta foi criada para sistemas operacionais Linux, apresentado um funcionamento limitado em Windows. Por ser focada em uso comercial, ela contém uma documentação e suporte melhores, mas devido à sua complexidade, ainda necessita um investimento maior de tempo para que possa ser usada.

2.4.4 Métricas Monitoradas

Aplicar a avaliação de desempenho requer a definição de quais critérios e métricas de desempenho serão usados. Esta definição deve ser iniciada por quais métricas são mais relevantes no experimento em questão. As métricas coletadas por um monitor podem ser classificadas em três categorias (JAIN, 1991).

- **Responsividade**: Tempo necessário para a aplicação apresentar uma resposta;
- **Produtividade**: Volume de dados que são fornecidos em uma resposta;
- **Utilização**: Quantidade de recursos de desempenho do sistema operacional que são usados pela aplicação sendo monitorada. Este projeto tem como foco o uso de métricas de utilização.

Monitores de software proveem formas de registrar como uma aplicação consome recursos de desempenho de acordo com as restrições da ferramenta utilizada e do sistema operacional em qual a aplicação é executada. Dentre as métricas possíveis, as mais comuns entre as ferramentas de monitoramento são: CPU, RAM (RODOLA, 2021), taxa de I/O e armazenamento (HD) (BERKELEY, 2021; VLADISHEV, 2021).

O uso de CPU pode ser medido para uma aplicação específica ou para todo o sistema operacional, sendo comum que ferramentas forneçam este dado como a porcentagem de uso em uma janela predefinida de tempo. Uso de RAM pode ser medido fornecendo a quantidade usada por uma aplicação ou o sistema todo, em uma escala de bytes. Métricas de I/O e armazenamento podem fornecer o número de acessos de leitura e escrita, volume de dados armazenados ou transmitidos e recebidos.

Ao apresentar a metodologia Tricorder, Montes (MONTES, 2019) utilizou as métricas percentual de CPU, consumo de memória RAM em bytes e número de operações de I/O. Porém, a metodologia Tricorder utiliza um componente genérico, chamado *Thermometer*, o qual permite a utilização de qualquer ferramenta de monitoramento por software sob o mesmo. Sendo assim, é possível utilizar quaisquer métricas suportadas por monitores, como armazenamento em disco, taxas de download e upload, e outras métricas que possam ser mais significativas em domínios diferentes dos apresentados pelo autor (MONTES, 2019).

2.5 Considerações Finais

Neste capítulo foram descritos os conceitos e as dificuldades de aplicações de Aprendizagem de Máquina, assim como conceitos de VV&T e Monitoração de Sistemas Computacionais. Aplicações de IA e de Aprendizagem de Máquina em específico apresentam dificuldades nos aspectos de garantia de qualidade devido à complexidade que é característica destes tipos de software. Assim, é necessário buscar novos recursos e instrumentos que auxiliem na qualidade destas aplicações.

Este trabalho tem como objetivo verificar a eficiência da metodologia Tricorder em aplicações de Aprendizagem de Máquina, que, caso seja comprovada, poderá ser um novo instrumento na realização de testes desse tipo de sistemas. Desta forma, os conceitos apresentados neste capítulo são essenciais para este estudo, e serão utilizados no decorrer de todo o projeto.

No próximo capítulo é apresentada detalhadamente a metodologia Tricorder, cuja eficiência é objeto de análise deste estudo, em específico para a verificação de defeitos na fase de manutenção de aplicações de Aprendizagem de Máquina.

CAPÍTULO

3

TRABALHOS RELACIONADOS

3.1 Considerações Iniciais

Neste capítulo é apresentado o levantamento bibliográfico realizado com o objetivo de encontrar trabalhos relevantes para os objetivos deste estudo. Os trabalhos listados a seguir são usados como fonte de conhecimento sobre testes de software, monitoração de desempenho e abordagens similares a metodologia Tricorder (MONTES, 2019), que é foco deste estudo.

O levantamento detalhado nesta seção é uma expansão do levantamento realizado por Thiago de Jesus Oliveira Durães, pertencente ao grupo do Laboratório de Sistemas Distribuídos e Programação Concorrente (LaSDPC) (LASDPC, 2021) do ICMC/USP, em 2020. Esta expansão foi realizada para buscar novos artigos de interesse que tenham sido publicados em 2020 e 2021.

3.2 Levantamento Bibliográfico

O levantamento bibliográfico desenvolvido neste trabalho foi inspirado no protocolo descrito por Petersen et al (PETERSEN; VAKKALANKA; KUZNIARZ, 2015). Este protocolo estabelece que devem ser definidas questões de pesquisa associadas aos objetivos do levantamento e uma *string* de busca para ser utilizada em repositórios selecionados, coletando a lista de todos artigos encontrados em um intervalo de tempo definido. A partir disso, são aplicados filtros nesta lista, de forma que restem apenas publicações que se enquadrem nas categorias desejadas de trabalhos publicados. Estas etapas são apresentadas a seguir.

Para a realização do levantamento bibliográfico foram definidas as seguintes questões de pesquisa (QP) para serem respondidas com a análise das publicações selecionadas:

- **QP1:** Quais trabalhos usam aspectos de desempenho para revelar defeitos em programas desenvolvidos para aprendizagem de máquina?

- **QP2:** Quais as principais métricas de desempenho usadas para caracterizar o uso de recursos computacionais?

- **QP3:** Como são feitas as validações das metodologias propostas para a revelação de defeitos por meio da análise de desempenho das aplicações?

O levantamento foi realizado com o uso das ferramentas de busca Scopus, IEEE XPlore e ACM Digital Library, utilizando a seguinte *string* de busca:

("Performance Analysis"OR "Performance Anomaly"OR "Performance Anomalies"OR "Performance Bugs"OR "Performance Bug"OR "Performance Objectives"OR "Performance Objective") AND ("Debugging"OR "Debug"OR "Clustering"OR "Application signature") AND NOT("medic"OR "image"OR "medical"OR "dataset")

Com o intuito de encontrar estudos recentes, foi utilizado um levantamento realizado previamente pelo grupo, com os mesmos critérios de busca, porém filtrado para encontrar trabalhos publicados entre os anos 2015 e 2019. Para complementar este levantamento inicial, foram realizadas outras duas buscas, a primeira em 2020 e a segunda no final em 2021. A primeira busca obteve 662 publicações, a segunda 239 e a terceira 6 novas publicações.

Para encontrar os trabalhos que pudessem ser mais interessantes para os objetivos deste estudo, as publicações encontradas pela primeira busca foram filtradas duas vezes. O primeiro filtro foi feito analisando os títulos e resumos, chegando ao número de 85 publicações. O segundo filtro foi realizado com base nos resumos, introduções e conclusões, assim localizando 30 itens de interesse.

Para a análise dos resultados da segunda busca, esta com somente os resultados do ano 2020, foram adicionados também todos os artigos que referenciam as 30 publicações extraídas da primeira busca, adicionando assim 97 publicações às 239 encontradas na busca. Estes 336 itens foram passados pelos mesmos filtros que a primeira busca, chegando a 12 artigos de interesse, totalizando 42 com os resultados anteriores.

Ao final de 2021 foi realizada uma nova extensão deste levantamento, buscando apenas trabalhos relevantes publicados no ano de 2021. Nesta terceira busca foram usados os mesmos critérios já utilizados nas duas buscas anteriores de artigos. Com isso foram encontradas mais 6 publicações, chegando no número de 48 publicações.

Estes 48 trabalhos foram classificados em grupos listados a seguir, com o intuito de facilitar a extração de informações. Estas publicações são apresentadas na Tabela 1.

Tabela 1 – Levantamento Bibliográfico - Publicações e Categorias

	Categorias							
Publicações	1	2	3	4	5	6	7	8
(LAABER; LEITNER, 2017)	-	-	-	-	X	X	-	X
(BANERJEE; SRIVASTAVA, 2019)	-	-	-	-	X	X	-	-
(WANG; JIN; YU, 2018)	-	-	X	-	-	-	-	-
(WANG *et al.*, 2017)	X	-	X	-	-	-	-	-
(MüHLBAUER; APEL; SIEGMUND, 2019)	X	-	-	-	-	X	-	X
(SILVA *et al.*, 2020)	X	-	-	-	-	X	-	X
(KHATUYA *et al.*, 2018)	X	-	-	-	-	X	X	X
(WERT; SCHULZ; HEGER, 2015)	-	-	-	-	X	-	-	-
(CASOLA *et al.*, 2017)	-	-	-	-	X	X	-	-
(KOU; CHEN, 2018)	X	-	-	X	-	X	X	-
(WURZENBERGER *et al.*, 2017)	-	-	X	-	-	-	X	-
(FOURNIER *et al.*, 2019)	-	-	X	-	-	-	X	-
(SHEN *et al.*, 2015)	X	X	-	-	-	-	X	-
(WIENKE; WREDE, 2016)	-	-	-	X	-	X	X	-
(SUJON *et al.*, 2016)	-	-	-	-	-	X	X	-
(SAUVANAUD *et al.*, 2015)	-	-	X	-	-	X	X	-
(MITRA *et al.*, 2015)	X	-	-	-	-	-	-	-
(ALAM; AHMAD; KO, 2017)	X	X	-	-	-	X	X	-
(WANG *et al.*, 2015)	X	-	X	-	-	X	X	-
(REICHELT; KüHNE, 2018)	-	-	-	-	-	X	X	X
(DAI *et al.*, 2019)	-	-	-	X	-	X	X	-
(SOUSA *et al.*, 2016)	X	X	-	-	-	X	-	-
(FAASSE; BUCEK; SCHMIDT, 2020)	-	-	-	-	X	X	-	-
(REICHELT; KüHNE; HASSELBRING, 2019)	-	-	-	-	-	X	X	X
(WEN *et al.*, 2016)	X	-	-	X	X	X	X	-
(DELGADO-PéREZ *et al.*, 2020)	X	-	-	-	-	X	X	-
(SáNCHEZ *et al.*, 2018)	X	-	-	-	-	X	X	-
(BHATTACHARYYA; AMZA, 2018)	X	X	-	-	-	-	X	X
(ARORA *et al.*, 2018)	-	-	-	X	X	X	-	-
(SáNCHEZ *et al.*, 2018)	X	-	-	X	X	X	X	-
(CHEN *et al.*, 2020b)	-	-	X	-	X	-	X	X
(GU *et al.*, 2017a)	X	-	-	-	X	X	-	-
(TIZPAZ-NIARI; CERNY; TRIVEDI, 2020a)	X	-	X	X	X	X	X	-
(LI *et al.*, 2019)	X	-	-	X	X	-	-	-
(ZHANG *et al.*, 2020)	-	-	-	-	X	-	-	-
(FU; PRIOR; KIM, 2019)	X	-	-	-	X	X	X	-
(GRECHANIK *et al.*, 2016)	X	X	-	-	X	X	-	-
(LUO *et al.*, 2016)	X	-	-	-	X	X	-	-
(LUO; POSHYVANYK; GRECHANIK, 2016)	X	-	-	-	X	X	X	-
(ISLAM; MANIVANNAN, 2017)	-	-	-	-	X	X	X	-
(JIMENEZ *et al.*, 2018)	X	X	X	-	X	X	X	-
(SáNCHEZ *et al.*, 2020)	-	X	-	X	-	-	-	-
(BELKHIRI; DAGENAIS, 2021)	X	X	-	-	X	-	X	X
(KHUDUR; POLOS, 2021)	-	X	X	-	X	X	-	-
(DELGADO-PéREZ *et al.*, 2021)	X	-	-	X	-	-	X	-
(LOMIO; JURVANSUU; TAIBI, 2021)	X	-	-	-	X	X	-	-
(MIHAYLOV *et al.*, 2021)	X	-	-	-	-	-	X	X
(CHANDRASHEKHAR; SANJAY, 2021)	X	X	-	-	X	-	-	X

1. **Caracterização de desempenho**: Demonstram formas de caracterizar o desempenho de um sistema sob teste. São de interesse para este trabalho pois a criação de perfis de desempenho é uma etapa essencial na metodologia estudada, podendo ser aprimorada com base nos conhecimentos extraídos das publicações deste grupo. O estudo realizado por Grechanik e Luo (GRECHANIK *et al.*, 2016), pode ser citado como exemplo, pois apresenta uma nova abordagem para definir um modelo de comportamento de aplicações a partir do monitoramento de dados de desempenho;

2. **Perfilamento**: Descrevem características de *performance* de um sistema a partir de perfis de desempenho. Estas publicações auxiliam no entendimento de tipos de problemas que podem ser identificados em sistemas a partir da interpretação de seus perfis de desempenho. Neste grupo pode-se destacar o trabalho de Sanchez et al (SáNCHEZ *et al.*, 2020) que apresenta uma taxonomia de *bugs* de desempenho, auxiliando na geração de estatísticas e melhor análise;

3. **Agrupamento de dados de desempenho**: Apresentam abordagens e ferramentas para agrupar dados de desempenho e extrair informações destes grupos. São interessantes para os fins deste estudo pois propõem técnicas diferentes da utilizada na metodologia Tricorder, permitindo a comparação entre elas. Se destaca o trabalho de Chen (CHEN *et al.*, 2020b) que utiliza as técnicas *Gaussian Mixed Model* e *Isolation Forest* para criar grupos de dados de desempenho;

4. **Causas de anomalias de desempenho**: Estes trabalhos apresentam formas de encontrar defeitos em sistemas de software que possam causar comportamentos distintos do esperado no desempenho. O conteúdo desses trabalhos é interessante, pois como a metodologia Tricorder tem o objetivo de identificar possíveis problemas a partir de dados de desempenho, esses estudos podem ser usados de forma complementar à análise, indicando as causas que possam ter causado as anomalias. Li et al (LI *et al.*, 2019), por exemplo, apresentam técnicas de aprendizado para identificar gargalos de desempenho e como estes podem ser causados por diferentes configurações do ambiente e da aplicação;

5. **Ferramentas de monitoração de desempenho**: Propõem novas ferramentas que têm como objetivo a monitoração de desempenho de sistemas. Como a metodologia Tricorder permite o uso de diversas ferramentas de monitoração, estas publicações são interessantes pois apresentam novas ferramentas que eventualmente podem ser usadas na metodologia. Neste grupo se destaca o trabalho de Gu (GU *et al.*, 2017a), que apresenta uma ferramenta que permite monitorar o uso de Processadores de Vídeo (GPU) por aplicações de *Deep Learning*;

6. **Análise de métricas de desempenho**: Apresentam meios de analisar dados de desempenho de sistemas. Estes trabalhos são interessantes para este estudo pois demonstram diferentes técnicas de análise de métricas de desempenho, permitindo a comparação de custo e eficiência entre elas e a utilizada na Tricorder. Um exemplo de trabalho deste grupo é o realizado por Luo et al (LUO *et al.*, 2016), que apresenta uma forma de analisar dados de desempenho utilizando Aprendizagem de Máquina, com o intuito de encontrar o melhor conjunto de entradas para realizar testes em aplicações;

7. **Detecção de anomalias**: Demonstram formas de se identificar anomalias de desempenho em aplicações. A detecção de anomalias de desempenho é uma parte essencial deste estudo, assim, estes trabalhos são ótimas fontes de conhecimento e comparação de eficiência. O trabalho de Islam e Manivannan (ISLAM; MANIVANNAN, 2017), por exemplo, apresenta uma técnica de monitoramento e detecção de anomalias de desempenho em aplicação *Cloud*;

8. **Uso de histórico de desempenho**: Demonstram técnicas e ferramentas que comparam dados de desempenho de versões anteriores com a versão atual. As abordagens apresentadas neste grupo podem ser utilizas para detectar regressões de desempenho em aplicações, e assim como a metodologia Tricorder, tem como foco a fase de manutenção de sistemas de software, podendo assim serem utilizadas para comparação. Luo e Poshyvanyk (LUO; POSHYVANYK; GRECHANIK, 2016) apresentam uma metodologia para detectar problemas de desempenho introduzidos em novas versões de um software a partir da comparação de dados de desempenho de versões anteriores e da versão atual.

Este conjunto de publicações permite responder às questões de pesquisa definidas previamente, analisando os artigos classificados nos grupos que possuem as informações requeridas pelas questões.

Analisando as publicações do grupo **detecção de anomalias** é possível responder a primeira questão de pesquisa. A maioria dos trabalhos deste grupo propõem a utilização de dados de desempenho para detectar apenas problemas de desempenho, como os estudos realizados por Wang et al (WANG *et al.*, 2017) e Kathuya et al (KHATUYA *et al.*, 2018). Poucas publicações foram encontradas que propõem a utilização de dados de desempenho para encontrar defeitos de maneira ampla, como o trabalho realizado por Fu, Prior e Kim (FU; PRIOR; KIM, 2019), que não focam somente na detecção de problemas de desempenho. Dentre estes, somente o trabalho de Tizpaz-Niari et al (TIZPAZ-NIARI; CERNY; TRIVEDI, 2020a) apresenta um uso em aplicações de Aprendizagem de Máquina, porém diferentemente da proposta deste trabalho de mestrado, a abordagem apresentada no artigo requer acesso ao código-fonte da aplicação sob análise.

Para a segunda questão de pesquisa é possível utilizar os artigos dos grupos **caracterização de desempenho** e **perfilamento**. Na maioria dos trabalhos foram utilizadas métricas

de uso de processador, memória, acessos de entrada e saída, e tempo de execução, como nas publicações de Silva et al (SILVA *et al.*, 2020) e Kou e Chen (KOU; CHEN, 2018), que usam estas quatro métricas, ou nos trabalhos de Wang et al (WANG *et al.*, 2015), Bhattacharyya e Amza (BHATTACHARYYA; AMZA, 2018) e Li et al (LI *et al.*, 2019) que utilizam um subconjunto destas métricas. Somente em dois artigos foram encontradas métricas que caracterizam o uso de recursos de aplicações de Aprendizagem de Máquina, o trabalho de Tizpaz-Niari et al (TIZPAZ-NIARI; CERNY; TRIVEDI, 2020a), que utiliza métricas de processador, memória e tempo de execução, e o trabalho de Gu et al (GU *et al.*, 2017a), que monitora *traces* de GPU para realizar a caracterização.

A terceira questão de pesquisa pode ser respondida analisando os artigos do grupo **detecção de anomalias**. Os trabalhos classificados neste grupo são validados com a realização de testes reais com bases de dados ou aplicações selecionadas pelos autores. É importante destacar que algumas das validações foram realizadas utilizando aplicações cujas fontes não são apresentadas, ou aplicações desenvolvidas pelos próprios autores, como é o caso dos artigos de Gu et al (GU *et al.*, 2017a) e Fu et al (FU; PRIOR; KIM, 2019), e por isso estão sujeitas a questionamentos sobre sua validade. Porém, neste grupo se destacam trabalhos que utilizaram dados abertos para validação, como os artigos de Islam e Manivannan (ISLAM; MANIVANNAN, 2017) e Chen et al (CHEN *et al.*, 2020b) que utilizam dados de monitoramento disponíveis na internet, e os trabalhos que utilizaram aplicações conhecidas ou de código-aberto como sistema sob análise, como os trabalhos de Tizpaz-Niari et al (TIZPAZ-NIARI; CERNY; TRIVEDI, 2020a) e Luo et al (LUO; POSHYVANYK; GRECHANIK, 2016), de forma que possam ser validados pelos leitores.

3.3 Considerações Finais

Neste capítulo foi apresentado o levantamento bibliográfico realizado para este estudo, o processo seguido para desenvolvê-lo e as publicações identificadas como relacionadas aos objetivos deste estudo. Os trabalhos selecionados foram classificados em oito grupos, conforme o levantamento realizado originalmente, porém com o incremento de dezoito novos artigos de interesse. Todas as publicações selecionadas apresentam informações interessantes para este estudo da metodologia Tricorder, porém dentre esses, alguns se destacaram por apresentarem metodologias ou ferramentas com abordagens similares à Tricorder.

O presente trabalho busca estudar o desempenho da metodologia Tricorder em sistemas de software de Aprendizagem de Máquina. Neste contexto não foram encontrados trabalhos com o mesmo objetivo de utilizar métricas de desempenho para testar aplicações de Aprendizagem de Máquina. Apesar disso, foram encontrados trabalhos que possam auxiliar em etapas do processo, como, por exemplo, na monitoração de aplicações de IA.

CAPÍTULO

4

METODOLOGIA TRICORDER

4.1 Considerações Iniciais

Este capítulo tem como objetivo apresentar e detalhar a metodologia **Tricorder**. São apresentados os principais conceitos da metodologia, uma demonstração em etapas de como ela pode ser executada na prática e um detalhamento do experimento realizado pelo autor da mesma com o intuito de testá-la e validá-la.

4.2 Definições da Metodologia

O trabalho desenvolvido por Montes (MONTES, 2019) propõe a metodologia Tricorder para a detecção de defeitos por meio do monitoramento dos recursos utilizados por um software sob teste. Dados de desempenho são coletados por meio da monitoração de um software e com estes dados é definido um perfil de uso que representa possíveis cargas de trabalho. Com o monitoramento ocorrendo em paralelo à execução, caso um perfil diferente seja identificado, isso pode ser considerado um indício da presença de um defeito.

A metodologia Tricorder consiste em monitorar o uso de recursos por um software considerado sem defeitos, já em seu ambiente de produção, armazenando um número N de amostras de dados. Estas amostras de dados são agrupadas de uma forma selecionada, sendo que no projeto original a forma de agrupamento selecionada foi o *Normalized Compression Distance* (NCD), seguindo a metodologia DAMICORE (SANCHES; CARDOSO; DELBEM, 2011). A metodologia DAMICORE realiza o agrupamento dos dados referentes ao uso de recursos do software de referência, criando um perfil de uso normal para tais recursos.

Após a definição do perfil de uso normal da aplicação, dados de uma nova versão do software, que pode ser proveniente de uma atualização ou manutenção, são adicionados gradativamente ao agrupamento. Caso um novo grupo seja criado com apenas dados da nova

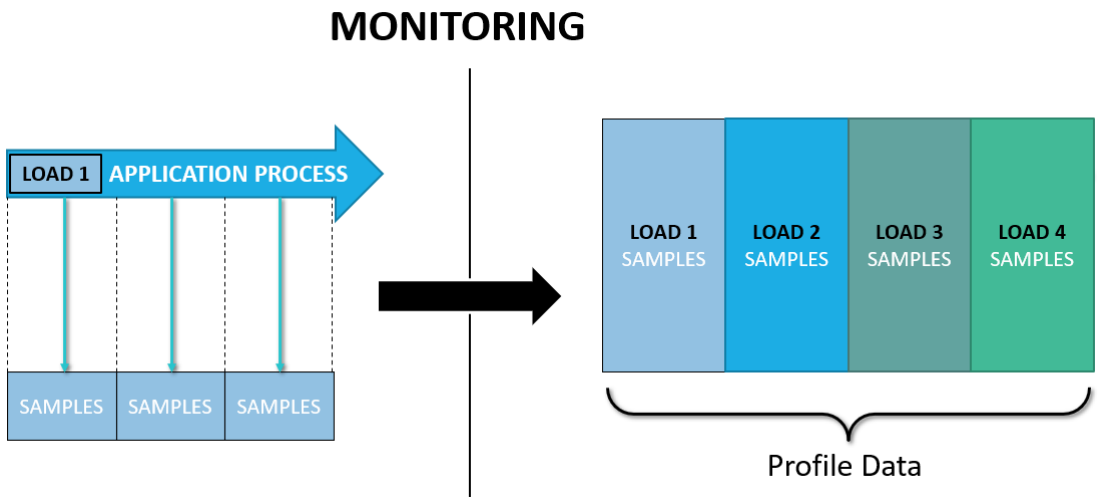
versão do software, isso é considerada uma anomalia pela Tricorder, em relação ao perfil de uso criado durante o agrupamento da aplicação original, indicando um possível defeito na nova versão do software, de acordo com a carga de trabalho onde foi acusada a anomalia. A anomalia indicada pela Tricorder é então analisada pelo testador que verificará se esta representa um novo comportamento diferente mas normal da aplicação, ou representa uma execução que revelou um possível defeito. Dessa forma, a Tricorder age como um alerta para o testador, que tomará a decisão final sobre a execução em teste.

4.3 Funcionamento

Para realizar a execução da metodologia Tricorder nos experimentos apresentados no projeto original (MONTES, 2019) foram desenvolvidos dois grupos de *scripts* Python, cujos nomes refletem a fase da metodologia a qual eles auxiliam: ***Monitoring*** e ***Analysis***.

A fase *Monitoring* tem a proposta de executar diferentes cargas de trabalho suportadas pela aplicação sendo testada, de forma a montar um perfil que represente todas as operações permitidas pelo software sob teste. Conforme ilustrado na Figura 3, a aplicação é executada múltiplas vezes, e durante cada execução são coletadas amostras de dados, a fim gerar dados para montar o perfil de desempenho.

Figura 3 – Tricorder - Fase *Monitoring*



Fonte: Elaborada pelo autor.

Nos experimentos originais esse processo foi automatizado, utilizando cargas definidas por diferentes arquivos JSON passados como argumentos para a aplicação. A aplicação é monitorada durante cada execução utilizando uma ferramenta desenvolvida por Montes (MONTES, 2019) chamada *Thermometer*, que utiliza a biblioteca PSutil do Python para monitorar o percentual de uso de CPU, a quantidade de bytes alocados na RAM e a quantidade de operações de I/O da aplicação e salvar em um arquivo com 300 amostras destes usos, retiradas a cada

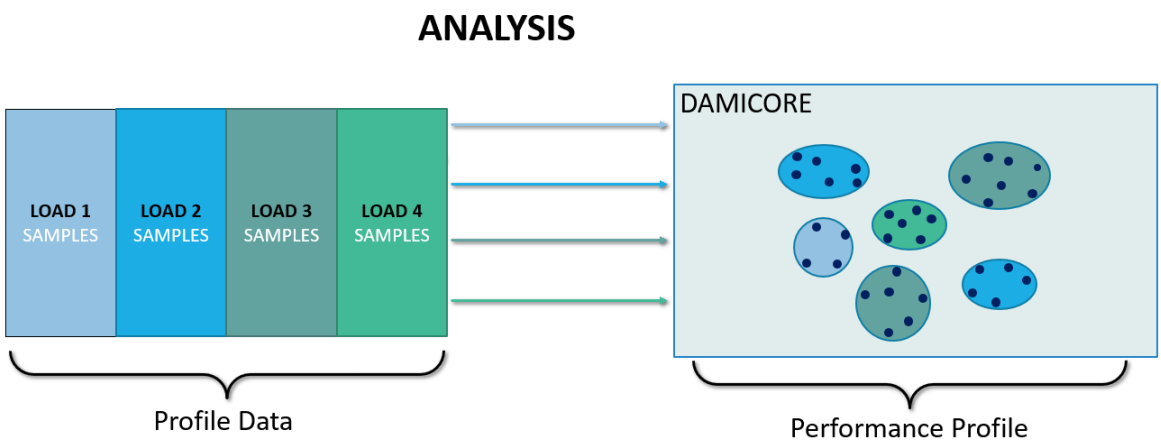
100ms da execução. Apesar da utilização destas três métricas de desempenho, é destacado que a metodologia suporta o uso de qualquer métrica, permitindo que sejam utilizadas mais ou até mesmo diferentes tipos de monitoramento que sejam mais adequados ao domínio a qual a aplicação sob teste pertence.

Para coletar dados suficientes para criar o perfil de desempenho é necessário que cada carga de trabalho seja executada múltiplas vezes. Para os experimentos executados originalmente, foi identificado empiricamente que o número de 30 execuções por carga de trabalho fornece dados suficientes para que a metodologia de agrupamento pudesse ser efetiva (MONTES, 2019). Um número maior de execuções é possível, mas o ganho de qualidade em relação ao tempo necessário não foi considerado como significativo neste caso. Este número de execuções não é fixo e pode ser alterado, podendo ser adequado de acordo com as especificidades do software sendo testado.

A fase *Monitoring* é concluída quando amostras de dados são salvas em arquivos representando cada carga de trabalho da versão original do software, considerada sem falhas, e das novas versões, que podem ser provenientes de manutenções ou atualizações. Todas as etapas descritas são realizadas utilizando *scripts* Python que automatizam o processo. Com todos esses dados armazenados é possível iniciar a fase de *Analysis*.

A fase de análise consiste no uso da metodologia DAMICORE (SANCHES; CARDOSO; DELBEM, 2011) para realizar o agrupamento de dados das amostras geradas na fase anterior. A Figura 4 ilustra como os dados coletados durante o monitoramento são fornecidos para a DAMICORE, que então realiza os agrupamentos, que passam a ser considerados como o perfil de uso da aplicação. Para permitir a automação desta etapa foi utilizada uma implementação em Python da metodologia, desenvolvida por Cesar (CESAR, 2016), chamada **Damicorepy**.

Figura 4 – Tricorder - Fase *Analysis*



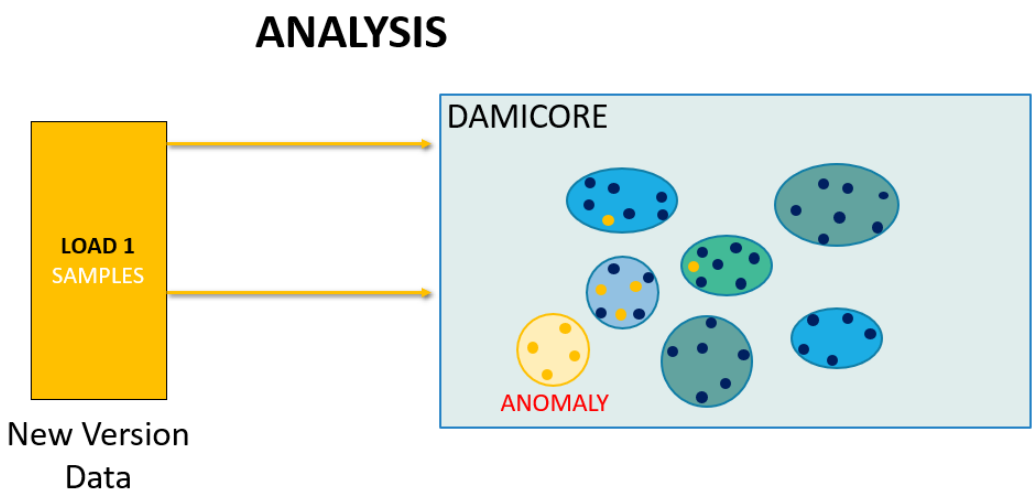
Fonte: Elaborada pelo autor.

Nesse passo é esperado que grupos sejam criados representando um perfil normal de uso da aplicação, considerando as diferentes cargas de trabalho utilizadas no monitoramento. Após

esse agrupamento inicial, gradativamente amostras de dados de uma nova versão da aplicação, ou seja, uma versão que sofreu uma manutenção ou atualização, são adicionadas a esta pasta e os agrupamentos são atualizados com estes novos dados, por meio do uso do Damicorepy. Isso se repete até que todos os arquivos de monitoramento, referentes às execuções de uma carga de trabalho, sejam agrupados ou até que o Damicorepy indique que um novo grupo foi criado somente com dados da versão com falha.

Quando um novo grupo é criado somente com dados da versão com falha, isso é considerado como uma anomalia e é levantada a possibilidade de existir uma falha na execução da carga de trabalho em questão. Este processo é ilustrado na Figura 5, onde alguns dos dados, indicados como pontos amarelos, foram adicionados aos agrupamentos existentes, simbolizando que estes foram considerados compatíveis ao perfil normal de uso. Porém, é criado também um grupo somente com dados da nova versão, pois estes dados não se encaixam dentro do perfil normal de uso, indicando a existência de uma anomalia.

Figura 5 – Tricorder - Fase *Analysis* - Anomalia



Fonte: Elaborada pelo autor.

Isso é realizado automaticamente para cada carga de trabalho de cada versão com falha, de forma que após a conclusão da fase *Analysis*, tenham sido criados registros das cargas de trabalho que apresentaram anomalias. Em termos práticos, a metodologia é aplicada da seguinte forma:

1. O programa de referência é executado N vezes para cada carga de trabalho suportada pela aplicação;
2. O programa sob teste é executado N vezes para uma carga de trabalho selecionada;
3. Todos os dados de monitoramento do programa de referência são agrupados gerando um perfil de uso normal;

4. Gradativamente arquivos de cada carga de trabalho do programa sob teste são agrupados;

5. Caso seja criado um agrupamento somente com dados da aplicação sob teste, isso é considerado como uma anomalia e um registro é salvo indicando um possível defeito na carga de trabalho sendo agrupada no momento;

6. Caso todos os arquivos de uma carga de trabalho sejam adicionados ao agrupamento e não seja criado um novo grupo, é assumido que nesta carga de trabalho não existe uma anomalia;

4.4 Validação Original da Metodologia

Para validar a metodologia, o autor desenvolveu um sistema cliente–servidor composto por três binários principais: aplicação cliente, aplicação servidor e biblioteca de processamento de dados (MONTES, 2019).

O sistema segue o seguinte fluxo de execução, conforme ilustrado na Figura 6:

1. A aplicação cliente é iniciada recebendo um arquivo JSON como referência, o qual define um ou mais arquivos dos tipos **BIN** e/ou **CSV** e um intervalo de tempo para cada arquivo;

2. O cliente estabelece uma conexão TCP com o servidor e entra em execução por tempo indeterminado, enviando os arquivos descritos no JSON em seus respectivos intervalos de tempo;

3. A cada pacote recebido, o servidor utiliza um objeto da classe ***Dispatcher*** para verificar se o envio foi concluído e, caso sim, enviar o pacote para ser processado pela biblioteca de processamento de dados;

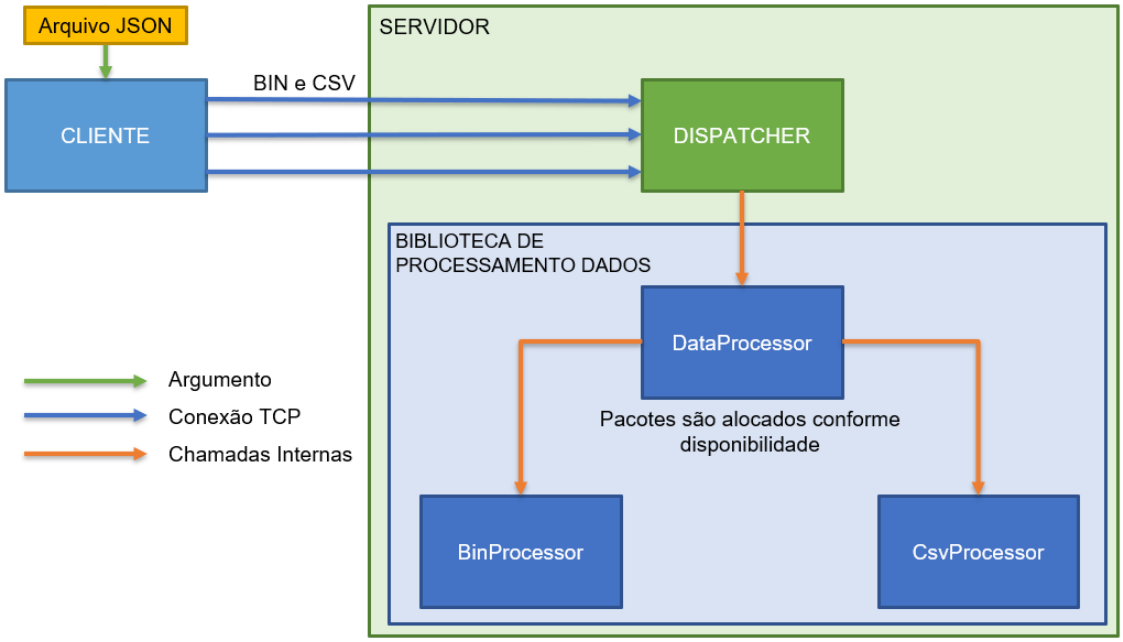
4. Na biblioteca os pacotes enviados são recebidos por um objeto da classe ***DataProcessor*** que utiliza uma fila para armazenar todos os pacotes recebidos pelo ***Dispatcher*** e utiliza quatro *threads*, dois para objetos ***BinProcessor*** e dois para ***CsvProcessor***, classes responsáveis por fazer o processamento de seus respectivos tipos de arquivo.

5. Conforme itens são adicionados na sua fila, o ***DataProcessor*** verifica se um dos objetos do tipo correto estão disponíveis, ou seja, não estão executando um trabalho e, caso não estejam, remove um pacote da fila e repassa para o processador correspondente.

Esse sistema, em sua versão original, é tratado como a aplicação de referência, a qual vai ser usada para criar o perfil de uso normal que considera todas as cargas de trabalho.

Para validar a metodologia foi proposta a criação de sete variações do sistema de referência, cada uma com a inserção de um defeito diferente. Os sete tipos de defeitos usados foram:

Figura 6 – Fluxo de Execução Cliente-Servidor



Fonte: Elaborada pelo autor.

1. ***BCLEAN***: Defeito prevenindo a limpeza de memória após seu uso;
2. ***INFINITE***: Defeito causando um *loop* infinito em um dos processadores de arquivos;
3. ***MONO***: Defeito reduzindo para um o número de *threads* usados para o processamento de arquivos;
4. ***SLEEP***: Defeito causando uma espera indevida;
5. ***SWAP***: Defeito que causa a leitura de uma estrutura diferente da enviada nos arquivos;
6. ***UNLOCK***: Defeito prevenindo a liberação dos *mutex* usados no uso de *threads*;
7. ***NOBREAK***: Defeito causando a execução de mais de uma opção em uma estrutura *switch case* do C++.

Essas versões com defeitos conhecidos têm o objetivo de serem usadas na metodologia Tricorder como uma nova versão do sistema, de forma que a metodologia possa ser validada quando for acusada uma anomalia nos agrupamentos causada pelos defeitos em questão.

Como o processamento real do sistema cliente-servidor é realizado pela biblioteca de processamento de dados, todos os sete defeitos propostos são inseridos em diferentes posições chave do código fonte da biblioteca. Com essas inserções o sistema chega a oito versões do sistema original, uma sendo a aplicação considerada sem defeitos, usada como referência e as demais sendo versões com falhas conhecidas.

Para a realização dos testes foram utilizadas nove entradas diferentes, divididas em três grupos, de acordo com o volume de suas cargas de trabalho: BCL, BCM e BCH, representando cargas BIN-CSV (BC) baixas (*Low*), médias (*Medium*) e altas (*High*), respectivamente.

Aplicando a metodologia sobre as sete versões com falhas, foram obtidos resultados positivos. Na Tabela 2 são apresentados os resultados obtidos, onde cada coluna representa uma das sete versões com falhas, e cada linha indica a carga de trabalho que foi utilizada. Nas células estão definidas quantas execuções do DAMICORE, dentre as 30 estipuladas anteriormente, foram necessárias até que a ferramenta acusasse a criação de um novo grupo e assim, a possibilidade de um defeito. As células preenchidas com a letra "N" indicam um falso negativo, sendo que no experimento em questão a ferramenta não identificou um novo grupo após ser aplicada, incrementalmente, sobre os dados de cada uma das 30 execuções.

Tabela 2 – Resultados do Experimento

CARGAS	BCLEAN	MONO	SLEEP	SWAP	INFINITE	NOBREAK	UNLOCK
BCL1	9	12	15	N	6	N	8
BCL2	4	5	17	N	5	N	14
BCL3	13	7	10	14	9	N	N
BCM1	8	9	6	15	7	N	4
BCM2	24	9	8	13	5	N	12
BCM3	8	7	12	22	5	N	13
BCH1	5	4	7	15	5	N	6
BCH2	6	9	7	26	5	N	3
BCH3	16	6	18	N	5	19	7

Fonte: Adaptada de Montes (2019).

Conforme os resultados apresentados, apesar de positivos na maioria dos casos, a metodologia Tricorder não foi capaz de detectar anomalias em experimentos com as cargas *SWAP*, *NOBREAK* e *UNLOCK*. De acordo com Montes (MONTES, 2019), os falsos negativos ocorreram em situações onde o defeito gera um impacto pequeno com a carga de trabalho utilizada, de forma que a metodologia DAMICORE não seja capaz de detectar uma anomalia. Isso ocorre principalmente nos experimentos com o defeito *NOBREAK*, pois este causa uma execução indevida de um código de processamento de arquivos BIN, que gera um pequeno impacto no desempenho, sendo detectado apenas com a carga de trabalho BCH3, de alto volume de processamento.

De forma geral, os resultados dos experimentos com a metodologia Tricorder se demonstraram favoráveis no contexto do trabalho realizado por Montes (MONTES, 2019). A metodologia foi capaz de identificar anomalias em todas as versões com defeitos em ao menos uma das cargas de trabalho, demonstrando que pode ser uma abordagem promissora, mas necessita de novos estudos.

4.5 Considerações Finais

Como apresentado nesse capítulo, a metodologia Tricorder teve bons resultados nos experimentos propostos pelo autor, demonstrando que ela pode ser um instrumento promissor na realização de testes de software, principalmente na fase de manutenção e suporte, e possivelmente em um fluxo de MLOps.

Por tratar o sistema sob teste como uma caixa-preta, a metodologia Tricorder cria a possibilidade de ser aplicada em diversas áreas, não sendo necessária a extração de características do software sob teste, de forma que não seja necessária instrumentação ou qualquer outra forma de interação no código fonte. Isso a torna particularmente interessante em contextos conhecidamente complexos, como no teste de software legado ou mesmo em aplicações de Aprendizagem de Máquina, estas o foco deste estudo.

CAPÍTULO

5

PLANEJAMENTO DE EXPERIMENTOS

5.1 Considerações Iniciais

Neste capítulo é detalhado o planejamento da execução dos experimentos com o objetivo de testar a metodologia Tricorder em aplicações de Aprendizagem de Máquina. São analisados o monitor e as métricas utilizados no primeiro experimento e novos candidatos que podem ser mais interessantes para os objetivos deste trabalho. São apresentadas também as análises realizadas para definir as aplicações e os defeitos a serem utilizados nos experimentos e a caracterização das cargas de trabalho de cada aplicação.

5.2 Seleção de Projetos

Com o objetivo de realizar experimentos com projetos que sejam representativos do campo de Aprendizagem de Máquina, foi decidido encontrar aplicações com características que as diferenciem, a fim de mostrar que a metodologia pode ser efetiva em diferentes contextos de Aprendizagem de Máquina. Para isto, foram seguidos os seguintes critérios de busca:

- Projetos com livre acesso e de código-aberto, permitindo verificação e reprodução dos experimentos;
- Projetos com número significativo de versões, apresentando um histórico real de falhas;
- Projetos com defeitos históricos já identificados e corrigidos, permitindo a utilização de versões com falhas e versões com estas falhas corrigidas;
- Projetos com características diferentes entre si, a fim de abranger diferentes contextos de Aprendizagem de Máquina;
- Projetos com tecnologias relevantes e populares, permitindo a demonstração da eficácia de metodologia em tecnologias utilizadas pelo público.

Após a definição dos critérios de busca, foram buscadas tecnologias e contextos populares. Esta etapa foi realizada com a ferramenta *Analyze Results* do *Web of Science* (SCIENCE, 2021), a fim de verificar as tendências do campo de Aprendizagem de Máquina na literatura desde o ano de 2015 até o ano 2021. A partir desta busca, foi identificado que a quantidade de estudos publicados de Aprendizagem de Máquina possui uma tendência positiva no intervalo de tempo definido. Mais especificamente, foram identificados três subcampos de Aprendizagem que também apresentam uma tendência positiva na quantidade de publicações:

- ***Deep Learning*** (DL);
- ***Reinforcement Learning*** (RL);
- ***Natural Language Processing*** (NLP).

Apesar desta tendência positiva, a quantidade de artigos de RL e NLP ainda é pequena em relação a DL, porém demostra aumento de interesse nestes subcampos. Buscando especificamente tecnologias e recursos de Aprendizagem de Máquina, foram identificadas as seguintes bibliotecas de Aprendizagem de Máquina utilizadas em um número crescente de publicações:

- **Scikit-Learn** (COURNAPEAU, 2021): Biblioteca possui recursos para Aprendizagem de Máquina;
- **TensorFlow** (BRAIN, 2021): Biblioteca possui recursos para diversos subcampos de Aprendizagem de Máquina, podendo ser utilizada com e sem GPU;
- **Keras** (CHOLLET, 2021): Biblioteca possui recursos para utilização de DL e NLP, apresentando também a variação Keras-RL, para aplicações de RL;
- **PyTorch** (FACEBOOK, 2021): Biblioteca possui recursos para diversos subcampos de Aprendizagem de Máquina, assim como TensorFlow.

Com a definição destas tecnologias foram então estabelecidas as seguintes questões: Quais aplicações serão utilizadas nos experimentos; quais cargas de trabalho serão utilizadas e quantas execuções; quais defeitos serão utilizados, e; onde estes defeitos deverão ser inseridos. Assim, o próximo passo realizado foi o de selecionar projetos para utilização nos experimentos.

Para selecionar as aplicações, foi utilizado o repositório GitHub (PRESTON-WERNER CHRIS WANSTRATH; CHACON, 2021). Neste foram filtrados os projetos mais populares que utilizam uma das bibliotecas listadas anteriormente. Além disso, foram ignorados projetos de *frameworks*, *wrappers*, bibliotecas e cursos, pois estes necessitariam de maiores intervenções externas para poderem ser utilizados como objetos de teste.

O primeiro projeto selecionado foi uma aplicação de *Deep Learning* capaz de gerar áudios a partir de uma entrada de texto e uma amostra de cinco segundos de voz, chamada de

Real-Time Voice Cloning (JEMINE, 2021). Esta aplicação utiliza a biblioteca PyTorch e, até o momento da escrita deste trabalho, possuía mais de 30.000 estrelas e 5.500 *forks* no GitHub.

O segundo projeto selecionado é chamado **BERT** (DEVLIN, 2020) e reúne diversos modelos para Processamento de Linguagens Naturais. Este projeto apresenta um novo modelo de representação de linguagem, que é pré-treinado e pode então ser aplicado para resolver problemas de Processamento de Linguagens Naturais (DEVLIN *et al.*, 2019). Ele foi desenvolvido utilizando TensorFlow, possuindo uma grande variedade de códigos e modelos prontos para NLP, apresentando cerca de 29.000 estrelas e 8.200 *forks*.

O terceiro projeto selecionado utiliza Aprendizagem com Reforço para reproduzir um jogo de xadrez, chamado ***Chess-Alpha Zero*** (PANG, 2020). Este projeto utiliza o algoritmo *AlphaGo Zero* proposto por Silver et al (SILVER *et al.*, 2017) para encontrar um modelo eficiente em partidas de xadrez. Esta aplicação utiliza Keras, e é menos popular que as demais, porém ainda assim se destaca dentre os projetos de Aprendizagem com Reforço, com 1.800 estrelas e 450 *forks* no GitHub.

Finalmente, foram analisados trabalhos de Aprendizagem de Máquina desenvolvidos dentro do programa de pós-graduação da USP ICMC (USP, 2021) a fim de selecionar um projeto com contexto diferente dos demais selecionados. Com isso, foi selecionado o trabalho de Santos e Silveira (SANTOS *et al.*, 2021), que utiliza a biblioteca scikit-learn para realizar testes com diversas bases de dados, como por exemplo a popular base de dados Iris (FISHER, 1936). Este trabalho se diferencia não somente pelo uso de uma biblioteca de Aprendizagem de Máquina diferente, mas também por se tratar de um *script* de Aprendizagem de Máquina tradicional, enquanto os demais selecionados atuam em subcampos específicos desta área de Inteligência Artificial. Este projeto é referenciado posteriormente neste documento como ***Decision-Tree***.

É importante destacar que o projeto de Santos e Silveira (SANTOS, 2021) foi desenvolvido antes deste trabalho, com outros objetivos. Ele foi selecionado com um dos projetos a serem utilizados neste trabalho pois se enquadra nos critérios definidos anteriormente, tendo sido desenvolvido sem qualquer interferência ou influência do trabalho atual.

Com isto foram selecionados projetos representativos em diferentes campos e tecnologias de Aprendizagem de Máquina. Todos os projetos foram desenvolvidos por terceiros e foram criados com objetivos diferentes, atendendo diferentes necessidades.

5.3 Avaliação de Métricas

Como a metodologia Tricorder permite o uso de diferentes métricas e monitores, e este trabalho tem como foco a análise da metodologia em aplicações de Aprendizagem de Máquina, diferentemente do estudo original, onde o objeto de estudo foi uma aplicação cliente-servidor tradicional, foi realizado um estudo avaliando outras possibilidades. Para validar as métricas

de desempenho utilizadas nos estudos anteriores e para buscar novas métricas que possam ser mais interessantes para aplicações de Aprendizagem de Máquina, foram realizadas três buscas na literatura.

Primeiramente foram revisados os trabalhos selecionados no levantamento bibliográfico realizado para este estudo e então foram extraídos artigos que tenham como foco análise ou monitoração de métricas de desempenho de aplicações de Aprendizagem de Máquina. Após isso foram verificados todos os artigos que referenciam os artigos selecionados na etapa anterior, a fim de procurar outros trabalhos com objetivos similares, que possam trazer métricas utilizadas para caracterização ou monitoração de aplicações de Aprendizagem de Máquina.

Com estas duas atividades, foram identificados 16 trabalhos que utilizavam ou mencionavam métricas de desempenho especificamente para o domínio de Aprendizagem de Máquina. Na Tabela 3 são apresentadas as métricas de desempenho identificadas e a quantidade de artigos em que elas foram utilizadas, e na Tabela 4 são apresentadas as métricas por publicação.

Tabela 3 – Métricas de Desempenho Encontradas

Métrica	**Abreviação**	**Quantidade de Artigos**
Uso de processador	CPU (%)	12
Quantidade de memória de vídeo	VRAM (MB)	10
Uso de processador gráfico	GPU (%)	9
Quantidade de memória	RAM (MB)	6
Acessos de entrada e saída	I/O	4
Carga de disco rígido	*Disk* (%)	2
Tempo de execução	*Time* (ms)	1

Tabela 4 – Métricas por Publicação

Publicações	**CPU**(%)	**VRAM**(MB)	**GPU**(%)	**RAM**(MB)	**I/O**	**DISK**(%)	**Time**(ms)
(GU *et al.*, 2017b)	-	-	X	-	-	-	-
(TIZPAZ-NIARI; CERNY; TRIVEDI, 2020b)	X	-	-	X	X	-	X
(IBRAHIM; WANG, 2020)	-	X	X	-	-	-	-
(JEON *et al.*, 2019)	X	X	X	X	-	-	-
(CHIEN *et al.*, 2020)	-	-	-	-	X	X	-
(CHEN *et al.*, 2020a)	X	X	-	X	-	-	-
(WANG *et al.*, 2019)	X	X	-	X	-	-	-
(CHOWDHURY *et al.*, 2019)	-	-	-	-	X	X	-
(DUBE; SURA, 2018)	X	X	X	-	-	-	-
(LI *et al.*, 2015)	X	X	X	-	-	-	-
(ZHANG; WANG; SHI, 2018)	X	-	-	-	-	-	-
(COLEMAN *et al.*, 2019)	X	X	X	-	-	-	-
(PREUVENEERS; TSINGENOPOULOS; JOOSEN, 2020)	X	X	X	X	-	-	-
(LIAO; CHEN; LI, 2020)	X	-	-	X	X	-	-
(ZINS; DAGENAIS, 2019)	X	X	X	-	-	-	-
(REN; YOO; HOISIE, 2019)	X	X	X	-	-	-	-

Analisando todas as métricas e extraindo as mais utilizadas, foi possível verificar que as três métricas utilizadas anteriormente: uso de processador, quantidade de memória e quantidade de entrada e saída, se demonstram interessantes também para Aprendizagem de Máquina. Nos estudos listados foi identificada uma variação em relação ao trabalho anterior, onde a métrica de entradas e saídas é utilizada de duas formas, quantidade de acessos e/ou quantidade de bytes de entrada e saída, assim ambas as formas foram selecionadas para serem utilizadas neste trabalho, devido a análises descritas posteriormente neste capítulo.

Além das três métricas originais, com o crescimento do campo de *Deep Learning* e mais especificamente, do uso de TensorFlow, duas novas métricas se destacaram: o uso de processador gráfico (GPU) e a quantidade de memória de vídeo (VRAM) utilizada. Com isto foi possível chegar a lista apresentada na Tabela 5, com as seis métricas julgadas mais interessantes para os objetivos deste trabalho.

Tabela 5 – Métricas de Desempenho Selecionadas

Tipo de Métrica	Componente	Descrição	Unidade
Existente	Processador	Carga em uso	%
Existente	Memória Primária	Quantidade	MB
Existente	Entrada/Saída	Acessos	Acessos
Nova	Entrada/Saída	Bytes	MB
Nova	Processador Gráfico	Carga em uso	%
Nova	Processador Gráfico	Quantidade Memória	MB

Conforme a Tabela 5, é possível verificar que dentre as métricas determinadas como mais interessantes para os objetivos deste trabalho, as três utilizadas em trabalhos anteriores se mantêm interessantes e utilizadas na literatura, com a adição da variação de bytes de entrada e saída.

Além destas, considerando os objetivos deste trabalho, foram adicionadas também duas métricas de uso de processador de vídeo, porcentagem de uso de processador gráfico e quantidade de memória de vídeo utilizada. Estas duas métricas demonstraram-se interessantes pois refletem o recurso principalmente usado em aplicações de *Deep Learning*, a GPU.

5.4 Avaliação de Monitores

Para avaliar novos monitores que pudessem ser candidatos para serem utilizados neste trabalho, foi realizada uma extensão da investigação de métricas. Com o levantamento bibliográfico realizado para avaliar métricas de desempenho, foram extraídos também quais monitores eram mencionados dentre os trabalhos selecionados.

Após este levantamento inicial, foram buscados monitores similares e concorrentes aos identificados na primeira busca. Com isso foram identificados nove monitores de interesse, apresentados na Tabela 6.

Tabela 6 – Monitores Encontrados

Tipo de Monitor	Nome
Infraestrutura	Ganglia
Infraestrutura	Zabbix
Infraestrutura	Nagios
Infraestrutura	Icinga
Infraestrutura	Centreon
Windows	*Performance Monitor* (Perfmon)
Linux	Strace / Ltrace / Ftrace
Linux	Perf
Linux	NvProf / Nvidia-SMI

Estes monitores podem ser classificados em três categorias principais: de Infraestrutura, projetados para monitorar diversos sistemas simultaneamente; de sistema operacional Windows, monitores nativos que podem coletar dados de processos e do sistema todo; e para distribuições Linux, que fornecem diferentes recursos de monitoramento para estes tipos de sistemas operacionais.

O primeiro monitor destacado é o **Ganglia** (BERKELEY, 2021), um dos monitores *open-source* mais conhecidos no mercado. É um suíte de software projetado para ser utilizado em *clusters* que utilizam sistema operacional Linux.

Este monitor é composto por duas partes principais: Gmond e Gmetad, onde Gmond é instalado em cada um dos nós, coletando a cada intervalo os dados do sistema, e Gmetad é instalado no nó principal. Em seu funcionamento básico, o Gmetad recebe as informações coletadas pelos Gmond instalados no nós e gera relatórios de forma a mostrar o desempenho de um *cluster* como um todo.

Após análise, foi possível verificar que o Ganglia é um monitor muito interessante para o contexto de *clusters*, fornecendo diversos recursos e, por ser *open-source*, permite acesso ao código-fonte e até mesmo eventuais adaptações no código, caso necessário.

Considerando os objetivos deste trabalho, alguns pontos podem ser destacados sobre o Ganglia: ele é projetado para sistemas Linux e para uso em *clusters*, limitando o seu leque de uso; sua configuração não é muito complexa, mas requer um pequeno investimento de tempo; permite monitorar GPU com o uso de *plugins*; permite a exportação de dados de monitoramento no formato CSV, e; até onde foi possível levantar, monitora nós inteiros, não sendo capaz de monitorar processos específicos.

O Ganglia poderia vir a ser bem interessante para monitorar aplicações para a Tricorder, caso esta metodologia fosse aplicada em *clusters*, ou em uma infraestrutura na qual o Ganglia já estivesse em uso. No entanto, neste momento, para monitoramento de processos específicos ou em diferentes contextos, o Ganglia não foi considerado como sendo a melhor opção.

O segundo monitor listado é o **Zabbix** (VLADISHEV, 2021), que também é um projeto *open-source*, focado em sistemas Linux. Diferentemente do Ganglia, cujo foco é *clusters*, o Zabbix pode ser usado em infraestruturas completas, com servidores, máquinas virtuais, equipamentos de redes e computadores convencionais.

Sua forma de coleta de dados é similar ao Ganglia, possuindo duas partes principais: os Agentes, instalados em todos os equipamentos que terão dados coletados; e um Servidor, que recebe os dados coletados pelos Agentes.

Apesar de ser um projeto de código aberto, o Zabbix teve um desenvolvimento mais focado na experiência do usuário, apresentando uma interface mais moderna e amigável. O Zabbix possui a limitação de ser projetado para sistemas operacionais Linux, porém pode ser usado em um leque maior de plataformas. A sua configuração é mais complexa que a do Ganglia, mas ainda assim é um processo intuitivo.

Considerando que este trabalho tem como foco o monitoramento de aplicações de Aprendizagem de Máquina, o Zabbix tem um ponto negativo, não possuindo um *plugin* nativo para monitorar GPU, dispondo apenas de extensões criadas por usuários. Em seu favor, ele permite o monitoramento de sistemas inteiros ou processos individuais, e dados podem ser exportados em CSV, facilitando o seu uso com a metodologia Tricorder.

É um monitor bastante amigável e flexível, podendo ser usado em diversos dispositivos, porém ainda é um software grande, sendo um monitor viável para contextos de infraestrutura, mas para monitoramento de um único processo ou software existem opções mais leves e objetivas.

Nagios (GALSTAD, 2021) é um conjunto de ferramentas de monitoramento e suporte para infraestrutura, dentre as quais, apenas o Nagios Core é uma opção grátis e de código aberto. Possui uma proposta similar ao Zabbix, podendo ser usado em vários equipamentos que compõe uma infraestrutura de TI.

Por possuir uma variedade de versões com propostas comerciais, a versão gratuita, Nagios Core, apresenta uma interface menos amigável e intuitiva. Assim como os monitores anteriores, tem como limitação sua compatibilidade apenas com sistemas Linux, e uma configuração mais trabalhosa e complexa, necessitando compilar o projeto para o instalar.

É uma ferramenta interessante, mas é menos polida e demanda um investimento de tempo maior do que o Zabbix para realizar sua configuração; porém permite monitoramento de GPU com o uso de *plugin* nativo.

O **Icinga** (ICINGA, 2021) também é um monitor de infraestrutura, cujo projeto é de

código-aberto. Diferentemente dos demais monitores, é compatível com sistemas Windows e Linux. Criado como um *fork* do Nagios, tem uma experiência bastante similar, porém com adaptações para ser mais compatível com sistemas operacionais Windows e uma interface diferente.

Por não ter uma proposta comercial como o Nagios, este software foi mais desenvolvido, oferecendo uma interface gráfica mais intuitiva e agradável. Por se tratar de um software grande, não é ideal para realizar monitoramentos rápidos ou de poucas aplicações, mas é uma opção interessante para monitoramento de infraestruturas.

Centreon (CENTREON, 2021), por fim, é o último monitor de infraestrutura listado. Possui uma proposta de configuração diferente, sendo instalado em um computador ou máquina virtual. O **Centreon** fornece todo um sistema operacional dedicado para a coleta de dados vindos de outros equipamentos.

O Centreon permite o monitoramento de diversos equipamentos por meio da instalação de *Appliances* em cada um deles, porém não utiliza um software instalado como Servidor; mas sim um sistema operacional modificado que é então acessado por outros computadores pelo navegador.

O Centreon permite a monitoração de sistemas operacionais Windows, e apresenta uma configuração mais objetiva, tendo seu servidor instalado como um sistema operacional. Porém, possui uma proposta comercial, assim como o Nagios, apresentando uma versão grátis pouco intuitiva, requerendo um investimento de tempo para alcançar uma configuração adequada. Como os demais monitores de infraestrutura, é uma opção interessante para monitorar sistemas inteiros, e aplicar a metodologia Tricorder em sistemas que já façam uso destes tipos de software.

Analisando monitores de sistemas operacionais, o ***Windows Performance Monitor***, ou **Perfmon** (MICROSOFT, 2021a), é uma ferramenta disponível em todos os sistemas operacionais Windows.

Ela permite o monitoramento do sistema ou de processos específicos, podendo ser utilizado por meio de uma interface gráfica, ou também de linha de comando, a tornando bastante flexível e facilmente adaptável para ser utilizada em *scripts*.

Com a introdução do PowerShell, um *shell* de linhas de comando disponível em sistemas operacionais Windows, o uso do Perfmon por linha de comandos se tornou ainda mais poderoso, possuindo uma grande variedade de recursos, como o comando *Get-Counter* que permite a utilização de *Performance Counters* (MICROSOFT, 2021b), ferramentas do Windows que fornecem uma camada de abstração, permitindo a coleta de diversos dados de uso de recursos de desempenho.

O Perfmon demonstra-se bastante promissor, sendo leve e flexível, podendo ser utilizado sem maiores esforços. A ferramenta é nativamente disponível em sistemas operacionais Windows, permitindo o monitoramento de diversos dados de equipamentos específicos ou processos, dentre

estes a GPU. Possui um grande foco no uso por linhas de comando, facilitando a criação de *scripts*.

Analisando as ferramentas disponíveis em sistemas operacionais Linux, destacam-se **STrace**, **FTrace**, e **LTrace** (LEVIN, 2021), que podem ser utilizadas por linhas de comando em terminais de sistemas operacionais Linux. Estas ferramentas têm como objetivo o monitoramento de chamadas e acessos de processos ou do sistema:

- **Strace**: Monitora as chamadas de sistemas (*system calls*) realizadas por um programa;
- **Ltrace**: Monitora as chamadas a bibliotecas realizadas por um programa;
- **Ftrace**: Monitora as chamadas a funções do *Kernel* do Linux, feitas por um programa.

Estas ferramentas acima são úteis em situações específicas, como para análise da interação de um programa com o sistema operacional, bibliotecas compartilhadas ou o *Kernel*, permitindo assim realizar atividades de *debug*, por exemplo, sem que seja necessário acesso ao código-fonte. Estas verificações não se enquadram no escopo deste estudo, assim estas ferramentas não foram consideradas adequadas para os objetivos deste trabalho.

A ferramenta **Perf** (LINUX, 2021) pode ser instalada em sistemas operacionais Linux, e pode ser utilizada por linhas de comando. É um monitor bastante poderoso, permitindo coleta de dados, gravação, comparação e realização de *benchmarks*. Por ser utilizada por linhas de comandos, pode ser utilizada com facilidade em *scripts*.

Permite o monitoramento de componentes e processos, porém não de GPU, necessitando que seja utilizada em conjunto com mais ferramentas. Assim como o Perfmon, o Perf apresenta diversos recursos de monitoramento para seu sistema operacional, tornando-o um monitor muito interessante para o uso com a Tricorder.

NvProf e **Nvidia-SMI** (NVIDIA, 2021), por fim, são ferramentas também para sistemas Linux, utilizadas por linhas de comando. Estas duas ferramentas são disponibilizadas pela NVIDIA, umas das principais fabricantes de processadores de vídeo.

Permitem a monitoração de uso de GPU e VRAM de sistemas todos ou de processos específicos. Possuem um uso similar ao Perf, podendo ser utilizadas junto a ele, permitindo a coleta de diversas métricas.

Todos os monitores listados anteriormente são válidos e apresentam pontos fortes, podendo ser utilizados como *Thermomether* na metodologia Tricorder. Apesar disso, analisando todas as alternativas, o **PSUtil** (RODOLA, 2021) se mantém como a opção mais interessante para os objetivos deste trabalho, fornecendo flexibilidade e facilidade de uso.

A biblioteca de Python é capaz de oferecer os mesmos benefícios do Windows Performance Monitor e dos monitores de recurso de sistemas operacionais Linux, em uma única

linguagem, permitindo a criação de *scripts* que possam ser utilizados em ambos os sistemas operacionais.

Uma limitação do PSUtil, no que se diz respeito a este estudo, é que este não provê formas de monitorar o uso de GPU, e a quantidade de VRAM em uso. Esta limitação pode ser facilmente corrigida por meio do uso de uma biblioteca adicional, também disponível para a linguagem Python, o **GPUtil** (MONRTENSEM, 2021). Esta biblioteca permite a abstração de recursos de monitoramento de diferentes sistemas operacionais, e é capaz de coletar dados de uso de placas de vídeo NVIDIA.

O GPUtil pode ser acrescentado com facilidade aos *scripts* já existentes da Tricorder, de forma a trabalhar em conjunto com PSUtil e coletar dados de processamento de vídeo, pertinentes para os objetivos deste trabalho. Utilizando as linhas de código apresentadas a seguir, é possível obter dados de uso de recursos gráficos.

```
1: gpu_load.append(GPUtil.getGPUs()[0].load * 100)
2: gpu_memory.append(GPUtil.getGPUs()[0].memoryUtil * GPUtil.
     getGPUs()[0].memoryTotal)
```

Considerando os pontos apresentados anteriormente, a ferramenta PSutil ainda se mantém a mais interessante e, quando utilizada junto ao GPUtil, mostra ser a opção mais simples e prática de se utilizar para os objetivos deste trabalho. Apesar disso, os demais monitores se mostram promissores, com pontos positivos em diferentes contextos, conforme apresentado na Tabela 7

Tabela 7 – Análise de Monitores

Monitor	Análise
Ganglia	Interessante para *clusters*
Zabbix	Interessante para grandes infraestruturas
Nagios	Versões comerciais mais interessantes
Icinga	Interessante para uso em infraestruturas com Windows e Linux
Centreon	Interessante para infraestruturas de TI que já o utilizam
Perfmon	Muito promissor, especialmente para projetos apenas Windows
Strace / Ltrace / Ftrace	Interessantes para análise de *traces*
Perf	Muito promissor, especialmente para projetos apenas Linux
NvProf / Nvidia-SMI	Muito promissor para uso com Perf

5.5 Avaliação de Compressores DAMICORE

Além de métricas e monitores, a metodologia Tricorder também permite a configuração de como são realizados seus agrupamentos. Mais especificamente, é possível alterar parâmetros de como a metodologia DAMICORE realiza estes agrupamentos.

A metodologia DAMICORE pode ser considerada como o coração da Tricorder, sendo responsável por criar o agrupamento dos dados de monitoramento, possibilitando a detecção de anomalias. No trabalho original, a DAMICORE foi utilizada como caixa-preta, sem que suas diferentes configurações fossem exploradas, porém, de acordo com o trabalho realizado por Bruno Kim (CESAR, 2016) a Damicorepy, biblioteca Python da metodologia, suporta diferentes configurações.

Dentre as configurações que podem ser alteradas no Damicorepy, pode-se destacar o tipo de compressor, pois este tem papel essencial no cálculo de NCD. Os experimentos realizados até o momento utilizaram o compressor ZLIB, porém, o Damicorepy suporta o uso de cinco compressores diferentes, apresentados na Tabela 8.

Tabela 8 – Compressores DAMICORE

Compressor	**Parâmetros Adicionais**
ZLIB	*Level*
BZ2	*Level*
BZIP2	*Level*
GZIP	*Level*
PPMD	*Level*

Realizando breves experimentos com diferentes tipos de compressor, foi possível verificar que essa mudança tem um impacto significativo. Os experimentos foram realizados utilizando cinco defeitos do tipo *SLEEP*, utilizado no trabalho original de Montes (MONTES, 2019), porém posicionados em pontos diferentes do código. Os resultados são apresentados na Tabela 9, onde são mostradas a média de iterações até a detecção, considerando somente execuções onde houve detecção de falha, o desvio padrão, e por fim, a taxa de detecção de falhas.

Tabela 9 – Experimentos com Compressores DAMICORE

Compressor	**Média de Interações (acertos)**	**Desvio Padrão**	**Taxa de Detecção**
ZLIB	8,51	7,22	86,67%
PPMD	7,00	8,00	84,44%
BZ2	7,54	7,91	82,22%
BZIP2	5,73	7,61	75,56%
GZIP	5,70	7,69	73,33%

Analisando os resultados apresentados na Tabela 9, é possível verificar que o ZLIB, compressor utilizado no estudo anterior, apresentou a maior taxa de detecção. Além deste, o PPMD, compressor padrão da ferramenta, mostrou-se interessante, com uma taxa de detecção menor do que o ZLIB, mas com uma média de iterações menor.

Os demais compressores tiveram resultados positivos, mas além de inferiores aos dois primeiros compressores, apresentam outro ponto negativo. Estes possuem um *overhead* maior, de forma que os compressores BZIP2 e GZIP chegaram a levar de duas a três vezes mais tempo que o ZLIB e o PPMD em cada uma de suas execuções. Cada execução com os compressores ZLIB e PPMD levaram, em média, 80 e 60 segundos, respectivamente, enquanto com os demais a média de tempo foi maior do que 200 segundos.

Por fim, o compressor ZLIB demonstrou-se como a melhor opção de compressor para realização de futuros experimentos, devido a seu tempo de execução e taxa de sucesso.

5.6 Padronização de Caracteres

Além dos diferentes parâmetros suportados pela metodologia Tricorder, foi realizada também uma investigação sobre a forma com que os dados de monitoramento são armazenados durante a fase *Monitoring*. Para buscar otimizar o uso da metodologia Tricorder, foi realizada uma investigação sobre o efeito do número de caracteres de cada parâmetro de monitoramento sobre a eficácia da metodologia. Em cada amostra coletada durante o monitoramento, os valores são adicionados a um arquivo CSV, o qual é utilizado posteriormente na fase *Analysis*.

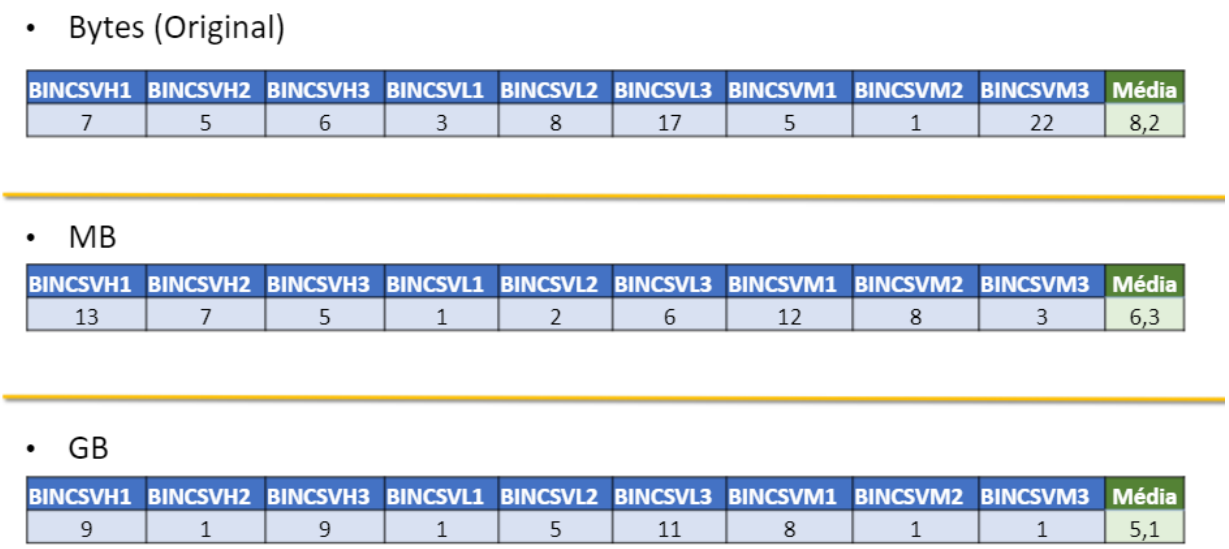
Nos experimentos originais, cada recurso monitorado foi armazenado em um arquivo CSV utilizando uma escala julgada propícia para o projeto em questão e seu uso de recursos. Porém analisando o arquivo CSV final, pode ser verificado que cada métrica apresenta um número médio de caracteres diferente das demais. Isso pode trazer um impacto ao resultado final, pois a metodologia DAMICORE, que realiza o cálculo de distância por meio da compressão, pode ser afetada por uma métrica com mais caracteres. Esta pode ter um peso maior do que uma métrica com menos caracteres.

Com o objetivo de investigar esta hipótese, uma base de dados de monitoramento foi analisada e modificada para verificar o impacto da padronização do número de caracteres. Na base em questão, a métrica CPU é armazenada em porcentagem, I/O em acessos e memória RAM em bytes, de forma que esta última apresentava uma quantidade média de caracteres muito maior que as demais. Para permitir a padronização, foi utilizado um *script* para transformar essa métrica de bytes para MB e GB, respectivamente.

Após a geração das novas versões em MB e GB, a fase de análise foi realizada novamente e a quantidade de execuções necessária para a identificação de anomalias foi comparada. Na Figura 7 são mostrados os resultados obtidos, sendo que em cada coluna são apresentadas as execuções de uma carga de trabalho diferente, e na direita é mostrada a média final.

Conforme a Figura 7, pode-se observar que, diminuindo os caracteres pela conversão para MB e GB, os resultados foram melhores necessitando de menos execuções para que uma anomalia fosse detectada. Com base nisso, nos experimentos realizados neste projeto, a métrica

Figura 7 – Padronização de Caracteres



Fonte: Elaborada pelo autor.

de uso de memória RAM será convertida para MB ou GB, conforme a média de uso do software sob teste, com o intuito de diminuir a diferença do número de caracteres usados em cada métrica analisada pela DAMICORE.

5.7 Comparação das métricas de Acessos e Bytes de I/O para ML

Conforme a análise apresentada anteriormente, métricas de I/O se demonstram interessantes no contexto de Aprendizagem de Máquina. Porém esta métrica pode ser utilizada de duas formas: número de acessos e bytes acessados. Com o objetivo de verificar qual das duas formas de I/O proporciona resultados melhores dentro do contexto de Aprendizagem de Máquina, foram realizados experimentos com as duas opções, separadas e em conjunto.

Na Figura 8 são apresentados os resultados obtidos quando utilizada cada configuração de métricas de I/O. Para realizar este experimento foram realizados novos monitoramentos, coletando ambas as métricas. Estes valores coletados foram processados e então foram criadas duas novas versões a partir destes, com todas as demais métricas, mas uma apenas com acessos e outra somente com bytes. Por fim pode-se verificar que diferentes métricas apresentaram resultados melhores em cada um dos projetos, porém considerando a média geral, a utilização de ambas as métricas trouxe o melhor resultado.

Desta forma, nos experimentos detalhados posteriormente neste trabalho, foram utilizadas ambas as métricas de Entrada e Saída: número de acessos e bytes acessados.

Figura 8 – Comparação Acessos e Bytes

- Acessos e Bytes (IO)

	BERT	*CHESS*	*VOICE*	Média
Execução 1	13,0	4,0	16,0	11,0
Execução 2	6,0	4,0	8,0	6,0
Média	**9,5**	4,0	12,0	**8,5**

- Acessos

	BERT	*CHESS*	*VOICE*	Média
Execução 1	11,0	3,0	12,0	8,7
Execução 2	27,0	3,0	8,0	12,7
Média	19,0	**3,0**	10,0	10,7

- Bytes

	BERT	*CHESS*	*VOICE*	Média
Execução 1	14,0	10,0	8,0	10,7
Execução 2	12,0	6,0	8,0	8,7
Média	13,0	8,0	**8,0**	9,7

Fonte: Elaborada pelo autor.

5.8 Experimentos Preliminares

Antes de iniciar a fase de injeção de falhas, foram realizados experimentos preliminares com o intuito de montar o ambiente necessário para executar cada umas das aplicações selecionadas. Todos os projetos possuem documentações detalhadas porém devido a lacunas de conhecimento foram necessárias maiores investigações.

Na máquina usada para os experimentos foram instalados Python 3.6.3, TensorFlow 1.15 e as versões mais atuais, até o momento de escrita deste documento, de PyTorch e scikit-learn. Nos experimentos preliminares foram preparados *scripts* para realizar 30 execuções, os valores monitorados tiveram seus números de caracteres padronizados conforme análise apresentada anteriormente. Para estes experimentos preliminares foi utilizada apenas uma carga de trabalho para cada aplicação.

A aplicação BERT foi testada em ambiente Windows, coletando apenas CPU, RAM e Acessos e Bytes de I/O. O defeito inserido foi um *SLEEP* de longa duração no início do código. A aplicação *Chess-Alpha Zero* foi testada usando as métricas de CPU, RAM, Acessos e Bytes de I/O, GPU e VRAM. A versão falha foi criada a partir da inserção de uma chamada *SLEEP* antes de cada jogada. A aplicação *Real-Time Voice Cloning* foi testada usando as métricas de CPU, RAM, Acessos e Bytes de I/O, GPU e VRAM, com a falha *SLEEP* inserida antes do processamento do arquivo de áudio de entrada.

A aplicação *Decision-Tree* foi executada em sistema operacional Ubuntu, coletando

apenas as métricas CPU, RAM e acessos de I/O. Para facilitar a realização de experimentos, esta aplicação foi adaptada para permitir a seleção de quais base de dados a processar, de acordo com os argumentos passados por linha de comando, e também a inserção da biblioteca Python *multithreading*, para permitir defeitos de concorrência. Foi criada uma versão com falha a partir da inserção de chamadas *SLEEP* antes da leitura de cada base de dados.

Com estes experimentos foi possível verificar que as aplicações eram executadas corretamente conforme os *scripts* criados. Nestes experimentos a metodologia Tricorder apresentou resultados positivos, sendo capaz de detectar as anomalias, porém devido ao uso de somente uma carga, e a inserção de falhas de forma *ad-hoc*, estes resultados positivos não foram considerados para os objetivos deste trabalho.

5.9 Busca de Falhas na Literatura

Com o objetivo de utilizar falhas realistas e verificáveis nos experimentos com a metodologia Tricorder, foram buscados na literatura artigos que apresentassem taxonomias de falhas de Aprendizagem de Máquina de forma geral ou de subcampos específicos desta área. Para realização deste trabalho foi utilizado o artigo de Humbatova (HUMBATOVA *et al.*, 2020) como base e a partir deste foi realizado *Forward* e *Backward Snowballing*.

Os artigos encontrados são apresentados na Tabela 10, sendo listados os artigos, as fontes a partir das quais foram encontradas as falhas e a quantidade de falhas identificadas. Na coluna Falhas são classificados como "Micro" as falhas quebradas em níveis mais baixos, abrangendo uma pequena variedade de problemas, e como "Macro" as falhas mais genéricas, abrangendo uma grande variedade de problemas. Estes artigos utilizam os sites GitHub (PRESTON-WERNER CHRIS WANSTRATH; CHACON, 2021), StackOverflow (SPOLSKY; ATWOOD, 2021) e entrevistas como fontes para encontrar falhas.

Tabela 10 – Publicações sobre Falhas de Aprendizagem de Máquina

Publicações	Fonte	Falhas
(HUMBATOVA *et al.*, 2020)	546 projetos analisados	93 tipos micro de falha
(WU; SHEN; CHEN, 2021)	146 programas analisados	4 tipos macro de falha
(JIA *et al.*, 2020)	300 correções de *bug* TensorFlow	11 tipos micro de falha
(JIA *et al.*, 2021)	300 *bugs* de TensorFlow	11 tipos micro de falha
(NIKANJAM *et al.*, 2021)	761 *bugs* de programas	11 tipos micro de falha
(THUNG *et al.*, 2012)	500 correções de *bug*	10 tipos macro de falha
(SUN *et al.*, 2017)	329 *bugs* de bibliotecas	15 tipos micro de falha
(KIM; KIM; LEE, 2021)	4577 *bugs*	7 tipos micro de falha
(CHEN *et al.*, 2021)	304 falhas	23 tipos micro de falha
(ISLAM *et al.*, 2019)	500 *bugs*	6 tipos macro de falha
(SHEN *et al.*, 2021)	603 *bugs*	12 tipos micro de falha
(ZHANG *et al.*, 2018)	175 *bugs* de TensorFlow	6 tipos macro de falha

Dentre as publicações encontradas, as mais interessantes são as que focam em fontes específicas. Por exemplo, foram selecionados somente projetos que utilizam as principais bibliotecas de Aprendizagem de Máquina, como é o caso de Humbatova et al (HUMBATOVA *et al.*, 2020) e artigos que analisam *bugs* encontrados dentro de bibliotecas de Aprendizagem de Máquina, como no estudo de Li et al (JIA *et al.*, 2020).

A busca na literatura foi encerrada quando os defeitos apresentados nas novas publicações podiam ser enquadrados nas mesmas categorias definidas nos artigos verificados anteriormente. Por exemplo, os artigos de Zhang et al (ZHANG *et al.*, 2018) e Chen et al (CHEN *et al.*, 2021) apresentam tipos semelhantes aos apresentados por Jia et al (JIA *et al.*, 2021) e Humbatova et al (HUMBATOVA *et al.*, 2020), respectivamente.

Em parte dos artigos analisados são apresentadas as quantidades de ocorrências de cada tipo de *bug*, dentro do domínio de cada publicação. Utilizando este recurso foi realizado o primeiro filtro para verificar quais as falhas mais comuns e que afetam o maior número de pessoas. A partir disso, foram selecionadas falhas com características distintas, com o objetivo de demonstrar a eficiência da metodologia para detectar diferentes tipos de falha.

No artigo de Humbatova et al(HUMBATOVA *et al.*, 2020) são apresentadas falhas exclusivas de aplicações de Aprendizagem de Máquina. Nesta publicação foram analisados somente projetos que utilizam as principais bibliotecas de Aprendizagem de Máquina, e não foram analisadas as bibliotecas em si. Os autores excluem defeitos mais abrangentes, que podem ocorrer em software de forma geral, e buscaram apresentar somente problemas exclusivos ao domínio de Aprendizagem de Máquina. Assim foi realizada uma pré-seleção dos tipos de falhas que se demonstram interessantes:

- **Modelo**: Estrutura de Rede Subótima - Falta ou excesso de camadas;
- **API**: Uso Errado de API - Falta ou chamada de API no local errado;
- **Treinamento**: Falta de Dados – Falta de colunas ou poucas linhas em uma base;
- ***Tensor***: Uso de *Shape* Errado de Tensor – *Shape* com dimensões errada.

No artigo de Nikanjam et al (NIKANJAM *et al.*, 2021) também são apresentadas falhas encontradas em projetos de Aprendizagem de Máquina que utilizam as principais bibliotecas, mas não foram analisadas as próprias bibliotecas. Nesta publicação foram identificados como interessantes os seguintes tipos:

- **Terminação**: Problemas para fechar/reiniciar ambiente – Chamada de *reset* no local errado;

- **Exploração**: Mal balanceamento entre exploração e execução – Parâmetros de exploração subótimos;
- **Modelo**: Problema de atualização de rede – Rede atualizada em passo ruim ou com parâmetro errado.

Finalmente, no artigo de Jia et al (JIA *et al.*, 2021) são apresentados tipos de falhas encontrados dentro das principais bibliotecas de Aprendizagem de Máquina. Estes defeitos são especialmente interessantes pois foram encontradas em bibliotecas amplamente utilizadas no campo e assim afetaram uma grande quantidade de desenvolvedores ao decorrer dos anos:

- **Tipo**: Uso de tipo diferente do adequado - Tamanho de variáveis;
- **Processamento**: Problema na inicialização de objetos ou uso de métodos - Falta de parâmetros;
- **Lógica**: Erros de lógica - Sequência errada de execução;
- **Configuração**: Problema de Configuração - Ausência de chamada de configuração;
- **Memória**: Problemas de uso de memória - *Memory Leak*;
- **Concorrência**: Problemas de sincronização - *Lock*.

Por fim, seguindo os critérios de encontrar defeitos que afetam a maior quantidade de usuários e também defeitos que abranjam contextos diferentes, foram selecionados sete tipos de falhas a serem utilizadas nos experimentos com a metodologia Tricorder:

- ***MODEL***: Problemas de configuração de Aprendizagem de Máquina (HUMBATOVA *et al.*, 2020; ZHANG *et al.*, 2018; NIKANJAM *et al.*, 2021; ISLAM *et al.*, 2019; SHEN *et al.*, 2015);
- **API**: Falta ou chamada de API no local errado (HUMBATOVA *et al.*, 2020; ZHANG *et al.*, 2018; SUN *et al.*, 2017; ISLAM *et al.*, 2019);
- ***TRAIN***: Falta de dados de treinamento (HUMBATOVA *et al.*, 2020);
- ***PROCESS***: Problema na inicialização de objetos ou uso de métodos (JIA *et al.*, 2021; THUNG *et al.*, 2012; SUN *et al.*, 2017);
- ***LOGIC***: Erros de lógica (JIA *et al.*, 2021; SUN *et al.*, 2017);
- ***MEMORY***: Problemas de uso de memória (JIA *et al.*, 2021; SUN *et al.*, 2017);
- ***CONC***: Problemas de sincronização (JIA *et al.*, 2021; SHEN *et al.*, 2015).

5.10 Definição de Falhas por Aplicação

Com a definição dos tipos de falhas que seriam utilizados neste projeto, de forma que estes tipos sejam embasados na literatura e baseados em dados reais, a próxima etapa realizada foi a de definir para cada projeto quais falhas específicas seriam inseridas. Considerando os critérios definidos anteriormente, para esta investigação foram utilizados os históricos de versões de cada um dos projetos, disponíveis no GitHub (PRESTON-WERNER CHRIS WANSTRATH; CHACON, 2021), de forma a encontrar falhas reais que possam ser enquadradas nas categorias definidas na sessão anterior.

Os códigos gerados com a injeção dos defeitos descritos nesta sessão, assim como os códigos de referência, considerados como sem defeitos, estão disponíveis no repositório deste trabalho (BRAGA, 2021). Neste repositório estão disponíveis também os dados de monitoramento utilizados ao decorrer deste projeto. Os dados de cada projeto estão organizados em diretórios com seus respectivos nomes, nos quais existe uma breve descrição dos arquivos contidos, assim como os arquivos auxiliares usados neste projeto, como os *scripts* utilizados para execução da metodologia Tricorder e para geração dos gráficos apresentados posteriormente.

Na aplicação BERT (DEVLIN, 2020), foram encontrados quatro defeitos históricos que se encaixavam nos tipos definidos. Assim três dos defeitos foram inseridos manualmente, conforme descrição encontrada na literatura:

- ***MODEL***: Injetado defeito do *Commit* a08ff, mudança de configuração com impacto grande, aumentando significativamente as execuções e curva de treino;
- **API**: Injetado defeito corrigido no *Pull Request* 69, uso de chamada de API errada causando impacto médio, pois força a utilização de métodos antigos de processamento de dados, que não possuem otimização;
- ***TRAIN***: Injetado defeito corrigido no *Pull Request* 137, uso de base de treino errada, sendo similar a base da carga SQuAD v1, causando impacto pequeno nesta e impacto grande na carga SQuAD v2;
- ***PROCESS***: Injetado defeito por uma mudança de argumento de método auxiliar, causando impacto médio, aumentando em 50% o número de camadas ocultas do TensorFlow, assim prolongando processamento;
- ***LOGIC***: Injetada pela mudança de fluxo de execução, com grande impacto pois inverte a execução das etapas de treino e teste;
- ***MEMORY***: Injetado defeito do *Commit* 2ad77b5, erro de alocação de memória de grande impacto, pois causa um grande volume de alocações desnecessárias;
- ***CONC***: Injetado defeito forçando que seja utilizada somente uma *thread*, causando grande impacto no uso de CPU.

Para a aplicação *Chess-Alpha Zero* (PANG, 2020) foram encontrados defeitos históricos dentro das categorias de defeito especificadas. Os defeitos utilizados neste projeto foram:

- ***MODEL***: Injetado defeito do *Pull Request* 5, mudança de configuração de impacto grande, forçando a utilização da configuração criada para servidores, ao invés da criada para processamento em computadores convencionais;
- **API**: Injetado defeito corrigido no *Pull Request* 90, valor de constante invertido, com impacto médio pois inverte a normalização dos valores usados no processamento;
- ***TRAIN***: Injetado defeito corrigido no *Commit* 42ea224, uso de base de treino errada alterando parâmetros iniciais de processamento, modificando as curvas de uso de recurso e causando impacto grande;
- ***PROCESS***: Injetado defeito corrigido no *Commit* da27793, valor errado passado como argumento causando impacto médio pois altera cálculo de uso de apenas um parâmetro, GPU;
- ***LOGIC***: Injetado defeito corrigido no *Pull Request* 8, mudança de fluxo de execução com impacto pequeno, pois causa o uso de um método antigo de processamento de rótulos, que é executado rapidamente no início do processamento;
- ***MEMORY***: Injetado defeito adaptado do *Commit* 11371a7, causando impacto médio ou grande, pois causa alocação desnecessária na memória, afetando mais cargas com maior utilização como *OPT* e *OPT-STEP*;
- ***CONC***: Injetado defeito corrigido no *Pull Request* 40, número de processos reduzido causando impacto médio, reduzindo uso de CPU durante o processamento.

Na aplicação *Real-Time Voice Cloning* (JEMINE, 2021), foi possível encontrar cinco defeitos adequados para os objetivos do trabalho. Desta forma, foi necessário que dois defeitos fossem inseridos manualmente, conforme descrição encontrada na literatura:

- ***MODEL***: Injetada por uma mudança na configuração do PyTorch, causando impacto médio, aumentando número de camadas usadas pelo PyTorch e taxa de aprendizado utilizada no treinamento;
- **API**: Injetado defeito corrigido no *Pull Request* 397, causa grande impacto pois faz com que a biblioteca use somente CPU, não usando GPU;
- ***TRAIN***: Injetada com o uso de arquivo de treinamento antigo, causando grande impacto pois requer um pré-processamento adicional, alterando curva de uso de recursos por todo o processamento;

- ***PROCESS***: Injetado defeito corrigido no *Pull Request* 678, causa impacto pequeno, pois faz com que possivelmente algum processo trave e seja reiniciado devido a uma divisão por zero causando exceção;
- ***LOGIC***: Injetado defeito corrigido no *Pull Request* 104, mudança de fluxo de execução com impacto pequeno, fazendo com que arquivo de áudio seja sempre normalizado, mesmo quando não necessário;
- ***MEMORY***: Injetado defeito do *Pull Request* 536, configuração com baixo uso de VRAM, causando impacto médio, pois faz com que processamento seja sempre otimizado considerando um sistema com pouca VRAM disponível;
- ***CONC***: Injetado defeito corrigido no *Pull Request* 838, redução de processos em sistemas Windows, reduzindo uso de CPU, causando impacto médio.

A aplicação *Decision-Tree* apresenta peculiaridades em relação às demais aplicações. Este software foi desenvolvido recentemente e não apresenta um grande número de versões, assim não foram utilizados defeitos históricos. Devido a esta limitação, as falhas inseridas neste projeto foram baseadas na literatura, em específico nos artigos apresentados anteriormente, de forma que estas falhas, apesar de não serem históricas, sejam injetadas a partir de taxonomias de defeitos baseadas em falhas reais. As falhas definidas para esta aplicação foram injetadas das seguintes formas:

- ***MODEL***: Mudança na configuração de KNN, causando impacto pequeno devido ao uso otimizado de recursos pela biblioteca Scikit-Learn;
- ***API***: Chamada de API no momento errado, causando processamento adicional, com impacto médio;
- ***TRAIN***: Leitura de poucos dados de treinamento, causando impacto médio, exceto para as cargas de trabalho *IRIS_HABER* e *WINE_HEART*, que utilizam as menores bases de dados;
- ***PROCESS***: Mudança de argumentos de métodos auxiliares, forçando o processamento de bases de dados diferentes das passadas como argumento, causando impacto grande;
- ***LOGIC***: Mudança de fluxo de execução em laço condicional utilizando para normalizar dados, arredondando para cima ou para baixo, causando impacto pequeno;
- ***MEMORY***: *Memory Leak* provocado pela injeção de variáveis com nomes duplicados em diferentes escopos. Causa impacto pequeno devido à quantidade de uso de recursos, especialmente nas cargas *IRIS_HABER* e *WINE_HEART*, que utilizam as menores bases de dados;

- ***CONC***: Remoção de *release* de um *mutex*, forçando o uso de apenas um *thread*, causando um impacto grande, exceto para a carga *OIL_MAMMO* que apresenta processamento mais rápido que as demais.

Além destes sete defeitos, nos experimentos são utilizadas também versões chamadas de ***CONTROL*** para cada uma das aplicações, contendo o mesmo código da versão original, de forma que este seja considerado sem defeito, e assim não deve ter as suas execuções consideradas como anomalias pela metodologia Tricorder.

5.11 Caracterização de Cargas de Trabalho

A partir da definição de quais projetos e quais falhas seriam utilizadas nos experimentos, foi realizado um estudo para definir e caracterizar as cargas de trabalho a serem usadas para cada projeto, de forma a contemplar o leque de operações de cada software.

O projeto BERT utiliza *benchmarks* encontrados na literatura para realizar experimentos e verificar a sua eficácia. Estes *benchmarks* utilizam bases de dados de Processamento de Linguagens Naturais, com dados e objetivos diferentes, de forma que a aplicação possa ser testada em diferentes contextos. Considerando a abrangência destes dados, as cargas de trabalho selecionadas para este software são as bases de dados que compõem os *benchmarks* GLUE (WANG, 2021) e SQuAD (RAJPURKAR, 2021). Tais cargas são processadas de formas diferentes no código, apresentando comportamentos diferentes.

Na Tabela 11 são apresentadas as cargas de trabalho, suas descrições, e seu uso médio de recursos, para cada uma das seis métricas estabelecidas anteriormente. Nas colunas referentes às métricas, dois símbolos são utilizados para indicar o comportamento: "^" utilizado para indicar picos nos valores indicados, e; "~" utilizado para indicar alta variação próxima ao valor indicado. As unidades utilizadas são: porcentagem para CPU e GPU; Megabytes (MB) para RAM, VRAM e I/O Bytes, e; número de acessos para I/O *Access*.

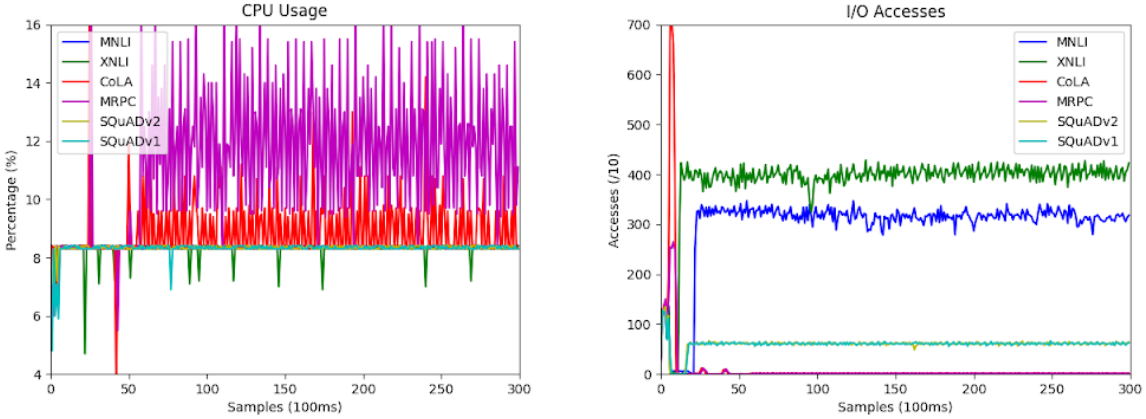
Tabela 11 – BERT - Cargas de Trabalho

CARGA	DESCRIÇÃO	CPU	GPU	RAM	VRAM	I/O AC.	I/O BYT.
MRPC (*Microsoft Research Paraphrase Corpus*)	Base de pares de frases, classificadas se semanticamente equivalentes.	~14,0%	~98,0%	3500	7900	^250 ~0	^400 ^450
CoLA (*Corpus of Linguistic Acceptability*)	Base de frases, classificadas se gramaticalmente corretas.	~9,5%	~98,0%	3500	7950	^700 ~0	^400 ^480
MNLI (*Multi-Genre Natural Language Inference*)	Base de pares de hipóteses, classificadas se validam ou contradizem a outra.	8,5%	~35,0%	1000	1000	~300	^35
XNLI (*Cross-Lingual Natural Language Inference*)	Parte da base MNLI, contendo 14 idiomas diferentes, porém menos dados.	~8,5%	^50,0%	500	1000	~400	^18
SQuAD v1 (*Stanford Question Answering Dataset*)	Base de questões, cujas respostas devem ser encontradas em bases de textos.	8,5%	^8,0% ~0,0%	480	1400	60	^30
SQuAD v2 (*Stanford Question Answering Dataset*)	Expansão da carga de trabalho SQuAD v1, com correções e mais dados.	8,5%	^8,0% ~0,0%	510	1250	60	^45

Fonte: Elaborada pelo autor.

Na Figura 9a é possível observar que as cargas MRPC, em roxo, e CoLA, em vermelho, apresentam maior uso de CPU que as demais cargas, e ambas têm uma grande variação no uso desta métrica. Apesar destas similaridades, a carga MRPC apresenta um uso médio de 14%, enquanto CoLA utiliza cerca de 9,5%. As demais cargas apresentam um uso menor e mais estável de CPU.

Ambas as cargas apresentam um pico de uso de acessos de I/O, como pode ser visto na Figura 9b, enquanto as demais cargas apresentam um uso constante. Comparando as duas cargas, apesar de apresentarem comportamentos semelhantes de acessos de I/O, a carga CoLA apresenta uma quantidade maior de acessos.



(a) CPU (b) Acessos de I/O

Figura 9 – BERT - CPU e Acessos de I/O

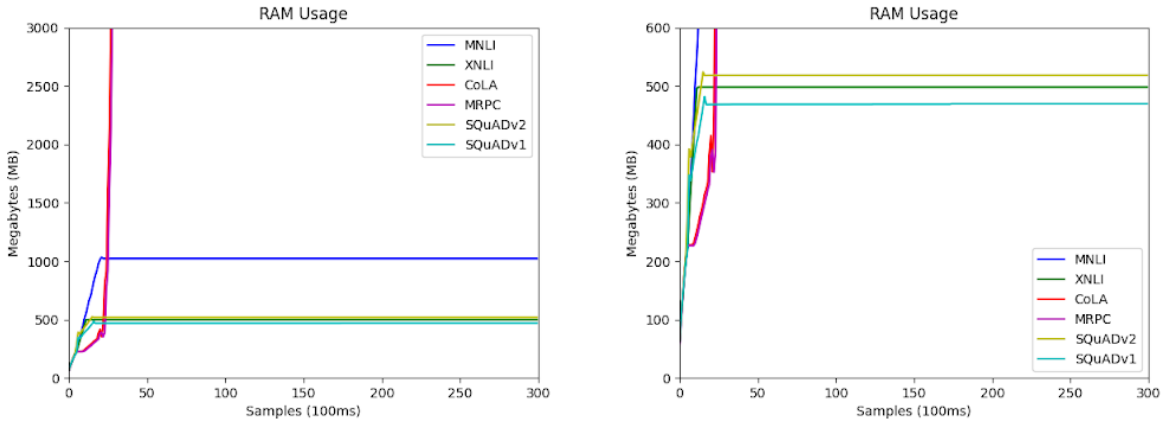
Fonte: Elaborada pelo autor.

Na Figura 10a pode-se observar que a carga MNLI, em azul, utiliza cerca do dobro de memória RAM da carga XNLI. Isso é esperado pois a base de dados XNLI, em verde, apesar de contemplar diversos idiomas, é composta apenas por um extrato da base MNLI, sendo assim menor em tamanho.

Analisando com uma escala ajustada o uso de RAM, apresentado na Figura 10b, pode-se observar que as cargas SQuAD apresentam uso muito próximo de memória, com a versão 2, em amarelo, apresentando um pouco mais, o que é esperado pois esta é uma atualização com correções e mais dados do que a versão 1, em ciano.

Analisando a Figura 11a, pode se observar que as cargas MRPC e CoLA apresentam um uso alto e constante de GPU, enquanto as cargas MNLI e XNLI apresentam um uso inconsistente. Analisando diversas execuções, pôde-se constatar que a forma de processamento das cargas MNLI e XNLI apresentam grande variação de uso de GPU entre execuções, enquanto as cargas MRPC e CoLA apresentam um uso consistente, próximo a 100%.

Apesar do seu uso maior de memória, devido às otimizações e por conter diferenças no

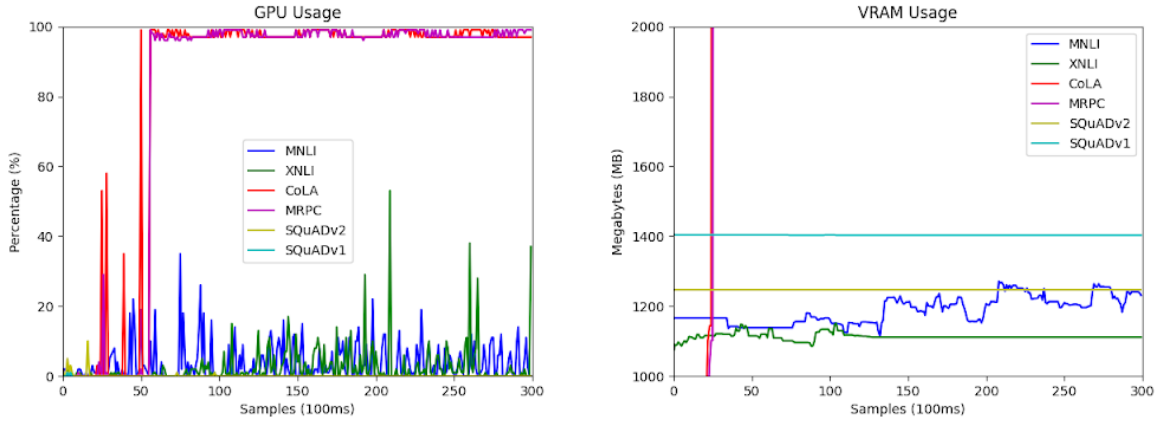


(a) RAM (b) RAM SQuAD

Figura 10 – BERT - Uso de memória RAM

Fonte: Elaborada pelo autor.

código de processamento dos dados, a versão 1 da base de dados SQuAD apresenta uso maior de VRAM, como pode ser observado na Figura 11b.



(a) GPU (b) VRAM

Figura 11 – BERT - Uso GPU e VRAM

Fonte: Elaborada pelo autor.

As cargas selecionadas para a aplicação BERT apresentam comportamentos distintos, contemplando os diferentes modos de funcionamento da aplicação. Além disso, as cargas selecionadas ativam diferentes partes do código, de forma que defeitos em partes específicas do código possam ser detectados.

A aplicação *Chess-Alpha Zero* apresenta uma diferença dentre os demais projetos quanto às suas cargas de trabalho. Este software não utiliza entradas convencionais para realizar suas operações, mas sim dispõe de cinco modos de operação, por meio dos quais é possível gerar novos modelos, analisar seus dados, comparar o modelo ótimo atual e então alcançar um novo

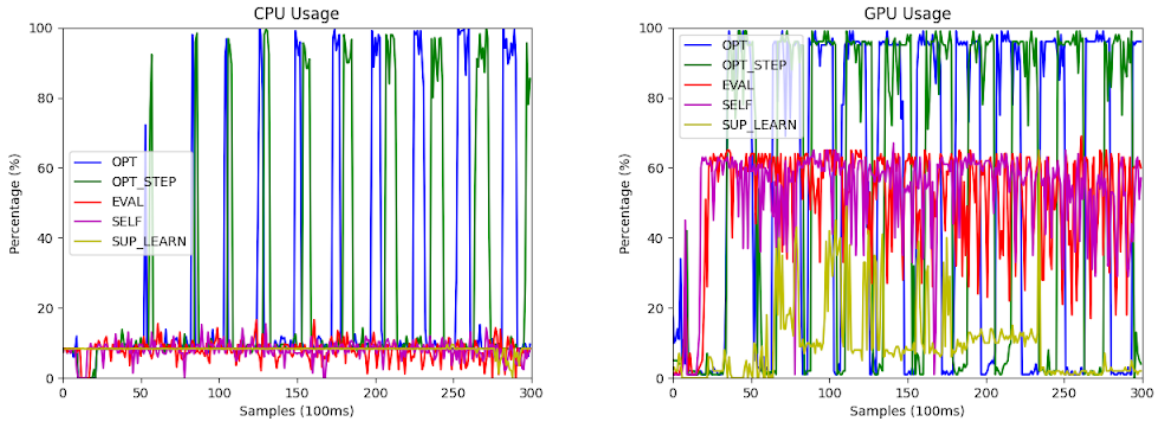
modelo mais eficiente. Por este motivo, as cargas de trabalho definidas para este projeto são todos os modos de operação permitidos por este software, conforme a Tabela 12. Esta tabela a mesma simbologia apresentada anteriormente para o projeto BERT, porém com a adição do símbolo ">", que indica um crescimento desde o valor da esquerda até o valor à direita do símbolo.

Tabela 12 – *Chess-Alpha Zero* - Cargas de Trabalho

CARGA	DESCRIÇÃO	CPU	GPU	RAM	VRAM	I/O AC.	I/O BYT.
OPT (*TRAIN*)	Tenta otimizar o modelo atual para encontrar modelos mais eficientes.	^~95,0%	^~95,0%	3000	8000	0	^~70
OPT_STEP (*TRAIN WITH STEPS*)	Mesmo que *OPT*, porém com número de passos (iterações de treino) maior.	^~95,0%	^~95,0%	3000	8000	0	^~70
EVAL (*EVALUATE BESTMODEL*)	Compara os modelos criados por *OPT* com o melhor modelo (*BestModel*) atual.	~10,0%	~60,0%	1500	8000	~300	^35
SELF (*SELF PLAY*)	Jogo automático de xadrez com melhor modelo para geração de dados para análise.	~10,0%	~60,0%	1500	8000	~200	^35
SUP_LEARN (*SUPERVISIONED LEARNING*)	Realiza um pré-treinamento do modelo com aprendizado supervisionado.	10,0%	^40,0% ~5,0%	0>4000	1000	^~800	^~25

Fonte: Elaborada pelo autor.

Estas cargas de trabalho têm propósitos diferentes, apresentando uso distinto de recursos. Nas Figuras 12a, 12b e 13a pode se observar que a carga *OPT*, em azul, e a carga *OPT_STEP*, em verde, apresentam um alto uso de CPU, GPU e Bytes de I/O em relação às demais cargas. Estas duas cargas apresentam comportamento similar, por realizarem a mesma função apenas com parâmetros diferentes, porém é possível verificar nas figuras que a carga *OPT* apresenta um intervalo menor entre picos.

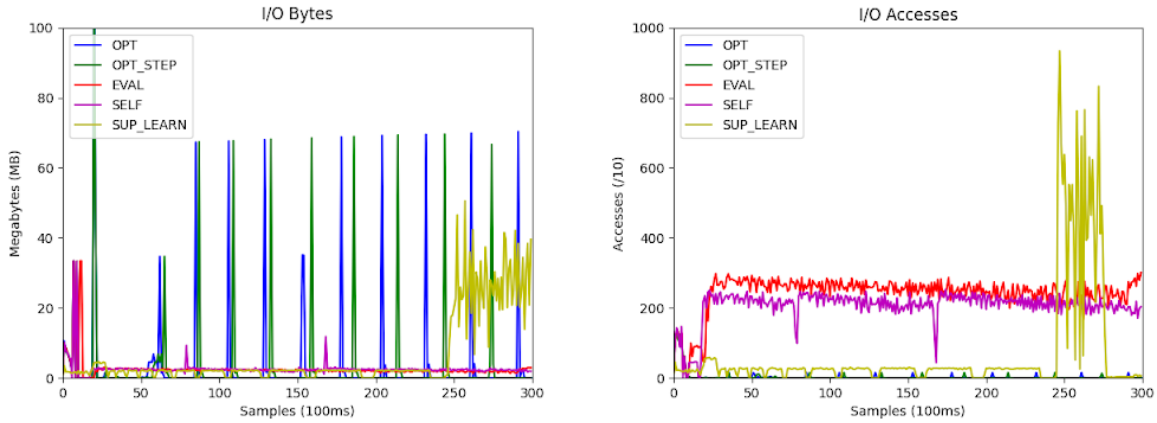


(a) CPU (b) GPU

Figura 12 – *Chess-Alpha Zero* - Uso de CPU e GPU

Fonte: Elaborada pelo autor.

As cargas *EVAL*, em vermelho, e *SELF*, em roxo, apresentam um uso médio de GPU, porém uma alta e frequente quantidade de acessos de I/O, como pode ser observado na Figura



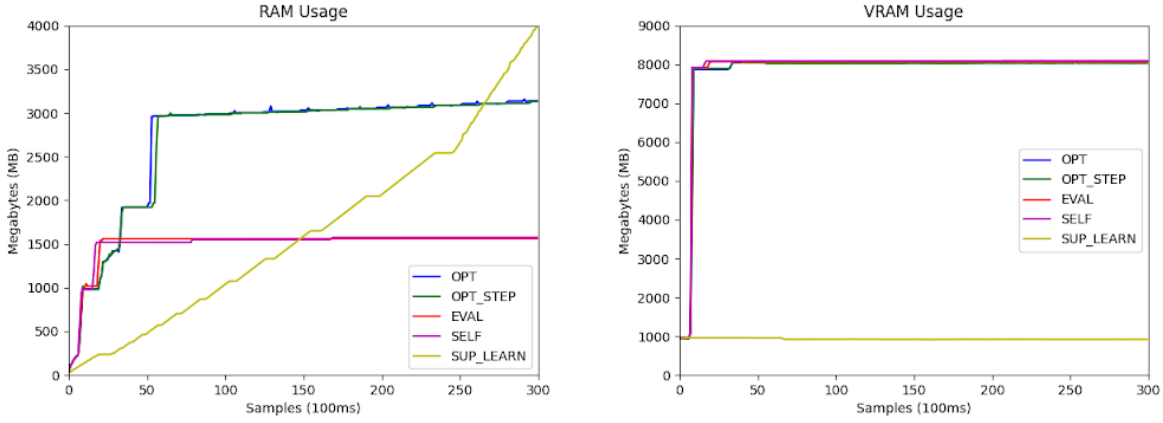
(a) Bytes I/O (b) Acessos I/O

Figura 13 – *Chess-Alpha Zero* - Bytes e Acessos de I/O

Fonte: Elaborada pelo autor.

13b. Ambas as cargas apresentam baixo uso de CPU. Analisando visualmente os gráficos de CPU e GPU não é clara uma diferença de uso destes recursos para estas duas cargas, mas analisando a quantidade de acessos de I/O é possível verificar uma tendência diferente entre as duas cargas.

Por fim, a carga *SUP_LEARN*, em amarelo, apresenta um comportamento distinto em relação às demais cargas. Analisando as Figuras 14a e 14b, pode-se verificar que esta carga de trabalho se comporta de forma diferente, apresentando um padrão único de curvas. Na Figura 13b é possível verificar que esta carga apresenta uma grande variação de acessos de I/O, a diferenciando das outras cargas de trabalho.



(a) RAM (b) VRAM

Figura 14 – *Chess-Alpha Zero* - RAM e VRAM

Fonte: Elaborada pelo autor.

Para a aplicação *Real-Time Voice Cloning*, como esta trabalha com duas entradas, um arquivo de áudio a ser clonado e um texto a ser gerado pelo software, foram definidos dois

arquivos de áudio e três textos de tamanhos diferentes, todos disponíveis ao público. Com estes dois grupos de entradas são formados então seis pares, que são utilizados como as cargas de trabalho deste projeto.

Os áudios selecionados são: um trecho de cinco segundos de uma palestra de George Takei (TAKEI, 2021) e; a leitura dos seis primeiros minutos do livro O Mágico de Oz (BAUM, 2021). Os textos usados como entrada são as 25, 100 e 1000 primeiras palavras do livro O Mágico de Oz (BAUM, 1900). Os dois arquivos de áudio e o livro podem ser encontrados gratuitamente para fins de teste, conforme suas referências. A lista de cargas de trabalho é apresentada na Tabela 13:

Tabela 13 – *Real-Time Voice Cloning* - Cargas de Trabalho

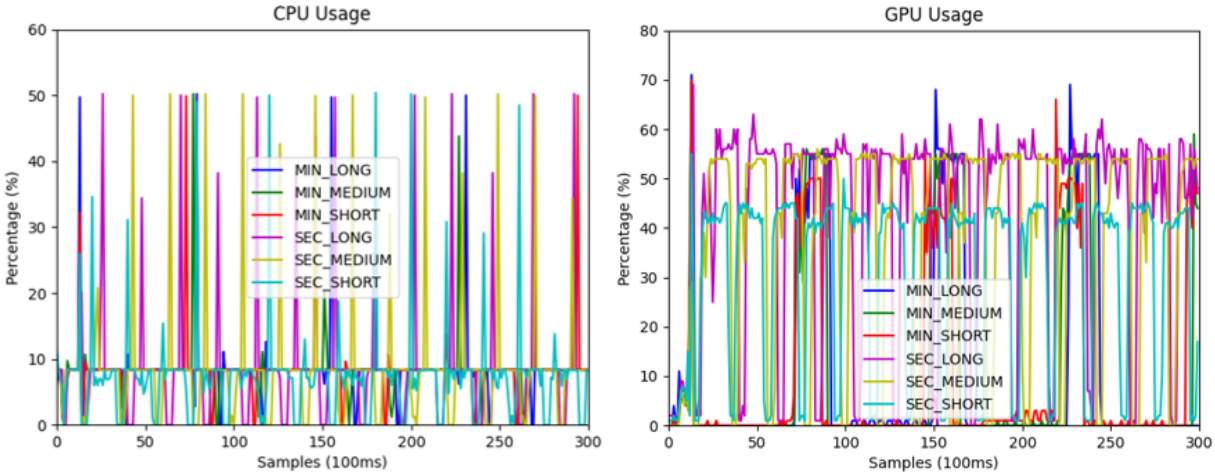
CARGA	DESCRIÇÃO	CPU	GPU	RAM	VRAM	I/O AC.	I/O BYT.
SEC_SHORT	Utiliza arquivo pequeno (60KB) para geração de áudio com poucas palavras (25).	^50,0%	55,0%	2000	3000	^15	^350
SEC_MEDIUM	Arquivo pequeno (60KB) para geração de áudio com quantidade média de palavras (100).	^50,0%	55,0%	2100	5000	^15	^350
SEC_LONG	Arquivo pequeno (60KB) para geração de áudio com muitas palavras (1000).	^50,0%	50,0%	2150	7500	^15	^350
MIN_SHORT	Arquivo grande (5MB) para geração de áudio com poucas palavras (25).	^50,0%	55,0%	~2200	3000	^1300	^50
MIN_MEDIUM	Arquivo grande (5MB) para geração de áudio com quantidade média de palavras (100).	^50,0%	55,0%	~2300	5000	^1300	^50
MIN_LONG	Arquivo grande (5MB) para geração de áudio com muitas palavras (1000).	^50,0%	50,0%	~2400	7500	^1300	^50

Fonte: Elaborada pelo autor.

Todas as cargas apresentam quantidade de uso similar de CPU e GPU, como pode ser verificado nas Figuras 15a e 15b. É possível verificar que os arquivos de entrada impactam o intervalo entre os picos, porém a quantidade de uso de recursos é similar, com apenas uma pequena redução no uso de GPU nas cargas com menos caracteres, *SEC_SHORT*, em ciano, e *MIN_SHORT*, em vermelho.

Analisando a Figura 16, pode-se verificar que diferentemente de CPU e GPU, todas as cargas apresentam usos distintos de VRAM. De acordo com o arquivo de áudio usado como entrada, a curva inicial de uso de recursos é alterada devido ao tempo necessário para o início do processamento, e a quantidade de caracteres altera a quantidade de VRAM em uso.

As Figuras 17a e 17b apresentam a quantidade de acessos de I/O em duas escalas diferentes. Analisando as figuras pode-se verificar que as cargas de trabalho que utilizam um mesmo arquivo de áudio apresentam comportamento similar entre si, porém entre os dois arquivos de áudio existe uma diferença significativa, onde as cargas *SEC* apresentam uma frequência de picos maior, porém uma quantidade de acessos significativamente menor do que as cargas *MIN*.

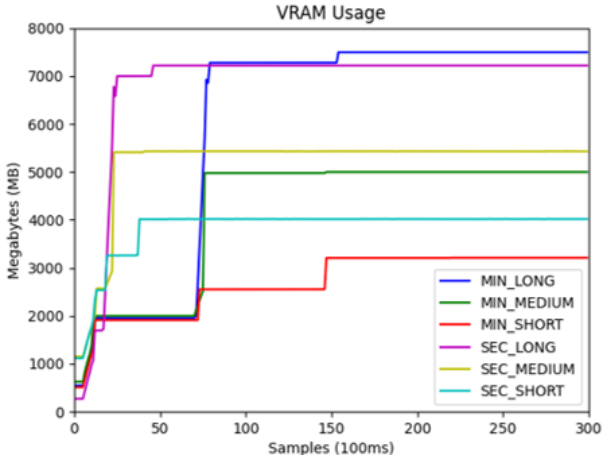


(a) CPU (b) GPU

Figura 15 – *Real-Time Voice Cloning* - CPU e GPU

Fonte: Elaborada pelo autor.

Figura 16 – *Real-Time Voice Cloning* - VRAM

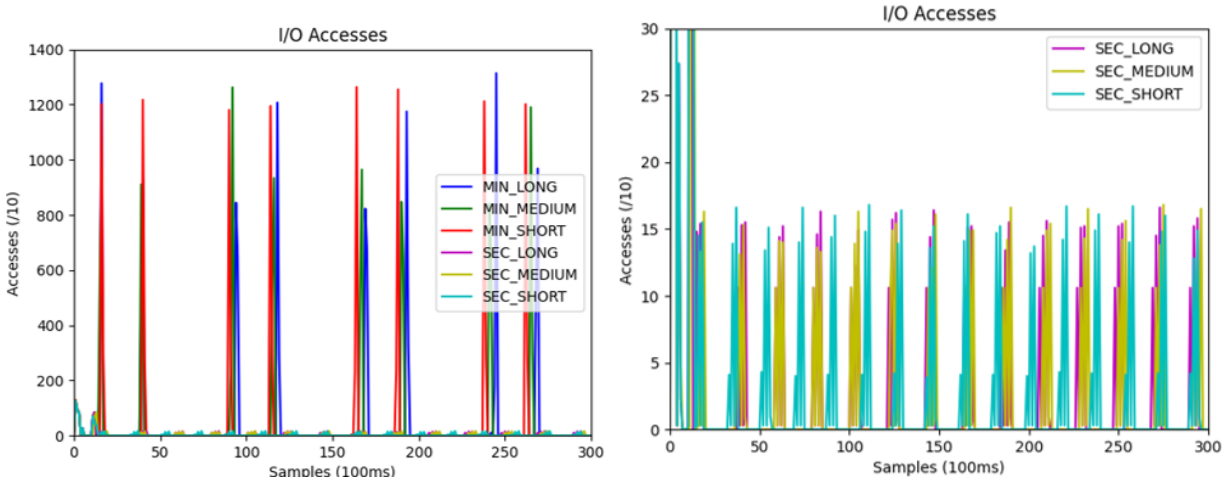


Fonte: Elaborada pelo autor.

Pode-se observar nas Figuras 18a que a quantidade de Bytes de I/O varia conforme o arquivo de entrada, com as cargas *SEC* apresentando um pico inicial e as cargas MIN apresentando picos menores mas recorrentes. Na Figura 18b pode-se verificar que as cargas apresentam uso de RAM crescente conforme a quantidade de caracteres de entrada. O uso de RAM apresenta uma relação com o uso de VRAM, com uma variação conforme o arquivo de entrada.

Analisando as cargas de trabalho da aplicação *Real-Time Voice Cloning* é possível verificar que apenas em algumas métricas é possível observar uma distinção significativa de uso de recursos entre as cargas de trabalho. De forma geral, esta aplicação apresenta diferenças discretas de uso de recursos entre suas cargas de trabalho.

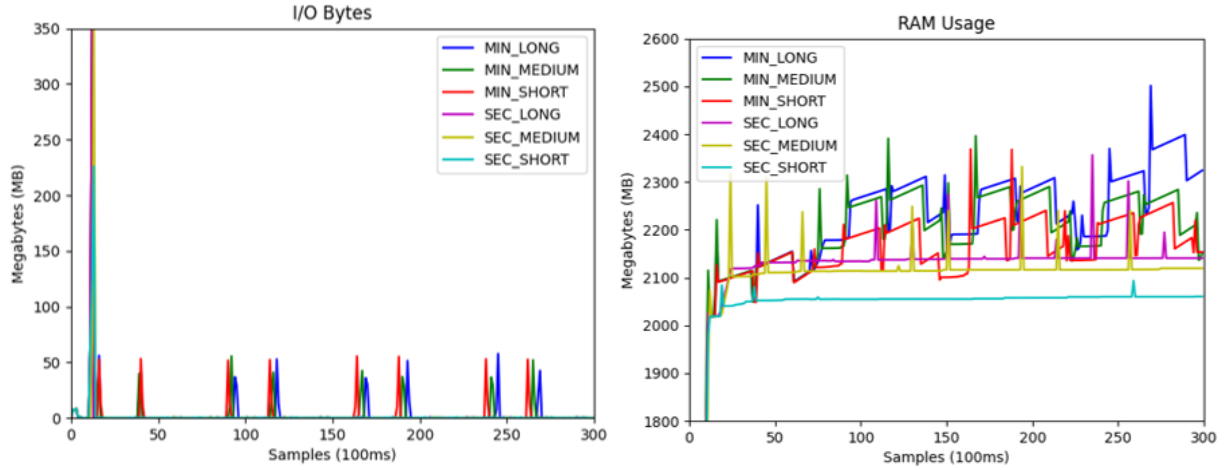
A aplicação *Decision-Tree* permite o processamento de doze bases de dados diferentes. Para padronizar a quantidade de cargas e permitir a execução por linha de comando, o código foi



(a) Escala Grande (b) Escala Pequena

Figura 17 – *Real-Time Voice Cloning* - Acessos de I/O

Fonte: Elaborada pelo autor.



(a) Bytes I/O (b) RAM

Figura 18 – *Real-Time Voice Cloning* - Bytes de I/O e RAM

Fonte: Elaborada pelo autor.

alterado para trabalhar com seis cargas de trabalho diferentes, cada uma processando duas bases de dados diferentes. Na Tabela 14 são apresentadas as cargas de trabalho deste software. Este projeto não utiliza GPU para realizar processamentos, assim, as métricas de GPU e VRAM não são utilizadas.

Tabela 14 – *Decision-Tree* - Cargas de Trabalho

CARGA	DESCRIÇÃO	CPU	RAM	I/O AC.	I/O BYT.
OIL_MAMMO	Utiliza bases de dados *Oil Spill* e *Mammography*.	^95,0%	^290	^600	^200
CANCER_PHON	Utiliza bases de dados *Breast Cancer* e *Phoneme*.	^90,0%	^300	^2500 ~500	^330
BANK_PIMA	Utiliza bases de dados *Banknote* e *Pima Indians Diabetes*.	^80,0%	^275	^1500	^250
IRIS_HABER	Utiliza bases de dados *Iris* e *Haberman Survival*.	^75,0%	^275	^1000	^250
IONO_SOLAR	Utiliza a base de dados *Ionosphere* e a base de dados *Sonar*.	^70,0%	^275	^1250	^225
WINE_HEART	Utiliza a base de dados *Wine* e a base de dados *Heart Disease*.	^50,0%	^275	^800	^250

Fonte: Elaborada pelo autor.

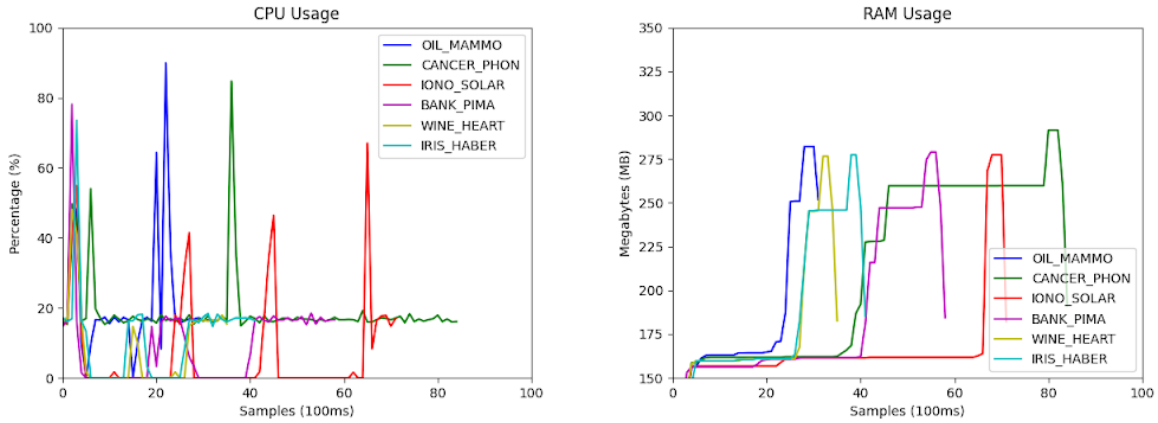
As cargas de trabalho da aplicação *Decision-Tree* não apresentam um padrão entre si, devido a variações no código utilizado para processar cada uma delas. Estas cargas apresentam variações no uso de recursos, duração e também início de processamento.

Analisando a Figura 19a é possível verificar que a carga *OIL_MAMMO*, em azul, apresenta o maior uso de CPU, porém apresenta um pico súbito, assim como as demais cargas. Esta carga é consistentemente a mais rápida a ser processada, enquanto a carga *CANCER_PHON*, em verde, apesar de ter o segundo maior uso de CPU, é consistentemente a mais lenta, levando o maior tempo para ser processada.

Na Figura 19a é possível observar que a carga *IONO_SOLAR*, em vermelho, apresenta um uso menor de CPU, porém realiza seu processamento em etapas, apresentando múltiplos picos de CPU. Diferentemente da carga BANK_PIMA, em roxo, que apresenta apenas um pico de uso de CPU no início de processamento e depois apresenta um uso contínuo mas baixo de CPU.

A carga *CANCER_PHON* apresenta um uso de memória RAM mais elevado que as demais cargas, com a carga *OIL_MAMMO* apresentando o segundo maior uso de RAM. A carga *IONO_SOLAR* apresenta um uso menor de RAM, com um pico de menor amplitude em relação às outras cargas.

Pode-se observar na Figura 20a que a carga *OIL_MAMMO* apresenta a menor quantidade de acessos de I/O. A carga *CANCER_PHON* apresenta um processamento mais lento em relação

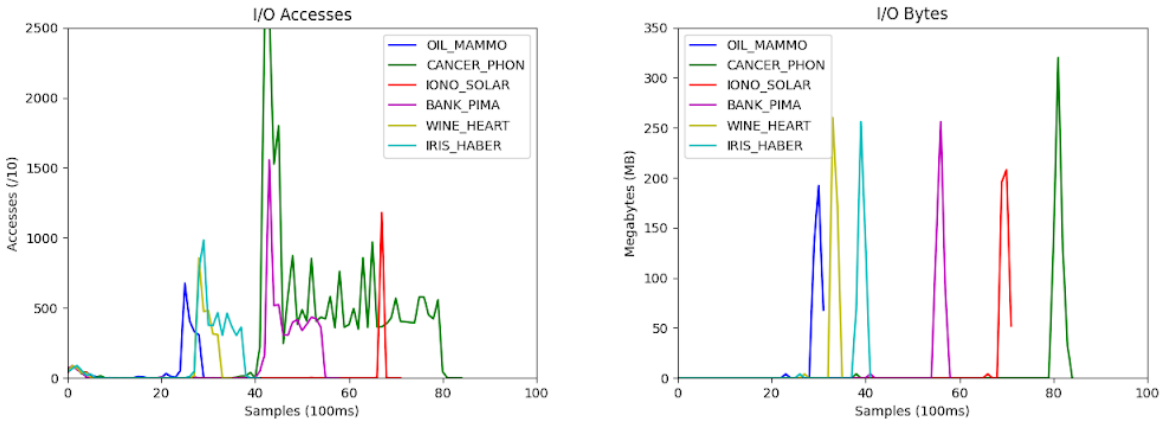


(a) CPU (b) RAM

Figura 19 – *Decision-Tree* - CPU e RAM

Fonte: Elaborada pelo autor.

às demais cargas, sendo possível verificar na figura que a curva de Acessos de I/O são mais prolongadas, assim como as curvas de uso de RAM, apresentadas anteriormente. A carga *BANK_-PIMA* apresenta uma grande quantidade de acessos de I/O, sendo o segundo maior uso, apesar de ter um pico e duração significativamente menor que a curva da carga *CANCER_PHON*.



(a) Acessos de I/O (b) Bytes de I/O

Figura 20 – *Decision-Tree* - Acessos e Bytes de I/O

Fonte: Elaborada pelo autor.

As cargas *WINE_HEART*, em amarelo, e *IRIS_HABER*, em ciano, apresentam comportamentos próximos, com poucas variações. Ambas as cargas são processadas rapidamente, seus perfis e tempos de execução podem ser observados na Figura 20b. A carga *IRIS_HABER* tende a apresentar um uso mais longo de RAM e acessos de I/O que a carga *WINE_HEART*, mas de forma geral ambas cargas têm comportamentos similares.

Dentre as quatro aplicações selecionadas, o software *Decision-Tree* apresenta o perfil de

uso menos definido, com grande variação entre as execuções, curta duração e curvas similares entre cargas de trabalho.

Conforme a caracterização apresentada nesta sessão, para cada um dos projetos foram definidas cargas de trabalho que demonstram os diferentes comportamentos contemplados em cada uma das aplicações selecionadas. Assim, ao se aplicar a Tricorder, espera-se que o perfil de desempenho criado pela metodologia seja representativo a todo o escopo de operação de cada aplicação. Cada um dos quatro projetos selecionados apresenta tipos de cargas e variações de uso de desempenho diferentes, de forma que seja possível verificar a eficácia da metodologia Tricorder em aplicações com usos e perfis de desempenho distintos.

5.12 Considerações Finais

A metodologia Tricorder suporta o uso de diferentes métricas, monitores, configurações da metodologia DAMICORE e até mesmo outras técnicas de agrupamento. Apesar desta gama de opções, as escolhas realizadas nos experimentos iniciais, conduzidos por Montes (MONTES, 2019), se mostraram válidas e dentre as melhores opções, considerando os objetivos das avaliações feitas. Assim, estas mesmas condições iniciais são utilizadas neste trabalho, apenas com a adição de novas métricas e o ajuste de caracteres de monitoramento.

Conforme as análises descritas neste capítulo, foi realizada uma investigação na literatura para definição de projetos e defeitos a serem utilizados neste trabalho. Para cada um dos projetos selecionados para serem objetos de teste neste estudo foram definidas cargas de trabalho distintas, dentre as limitações de cada projeto, com o objetivo de demonstrar a eficiência da metodologia Tricorder em cada um dos perfis de desempenho criados a partir das cargas de trabalho estabelecidas.

Com a definição das cargas de trabalho, de forma a criar um perfil de uso que contemple a operação normal de cada uma das aplicações selecionadas, os experimentos foram realizados e seus resultados são apresentados no Capítulo 6.

CAPÍTULO

6

RESULTADOS E DISCUSSÃO

6.1 Considerações Iniciais

Neste capítulo são apresentados e discutidos os resultados obtidos conforme o planejamento estabelecido no Capítulo 5. Os experimentos descritos neste capítulo foram realizados em um computador convencional com 16GB de memória RAM DDR4 e processador AMD Ryzen 5 5600X, com 6 *cores* e 12 *threads* e 3.70GHz de frequência. A placa gráfica utilizada foi a Nvidia Geforce GTX 1070 Ti, com 8GB de VRAM.

6.2 BERT - Resultados e Discussão

Os resultados obtidos aplicando a metodologia Tricorder no software BERT são apresentados na Tabela 15. Devido às diferenças de código para o processamento das bases de dados, os defeitos selecionados para injeção não afetam todas as cargas utilizadas, pois algumas cargas não permitem ativar os trechos de código em que os defeitos foram inseridos. Isso pode ser verificado com as cargas SQuAD v1 e SQuAD v2, as quais permitem que apenas os defeitos *MEMORY*, *MODEL* e *TRAIN* sejam reconhecidos nas execuções; os demais defeitos não são aplicáveis nestas duas cargas.

As células da Tabela 15 referentes às cargas não afetadas por defeitos são demarcadas com "N/A"(Não Aplicáveis). Na coluna *CONTROL*, as células marcadas com "OK" referem-se aos experimentos onde foram adicionados todos os dados de monitoramento e a metodologia Tricorder, corretamente, não detectou nenhuma anomalia. A coluna e linha ACERTO apresentam a porcentagem de acerto de cada uma das cargas de trabalho e defeitos, respectivamente. A célula no canto inferior direito da tabela, em cor diferente, apresenta a porcentagem total de acertos do experimento.

Tabela 15 – Resultados BERT

CARGAS	API	*CONC*	*LOGIC*	*MEMORY*	*MODEL*	*PROCESS*	*TRAIN*	*CONTROL*	ACERTO
MRPC	6	8	7	N/A	N/A	6	N/A	OK	100,0%
CoLA	11	10	6	N/A	N/A	20	N/A	OK	100,0%
MNLI	11	9	3	N/A	N/A	4	N/A	OK	100,0%
XNLI	7	13	3	N/A	N/A	11	N/A	OK	100,0%
SQuAD v1	N/A	N/A	N/A	5	7	N/A	23	OK	100,0%
SQuAD v2	N/A	N/A	N/A	6	4	N/A	3	OK	100,0%
ACERTO	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%

Fonte: Elaborada pelo autor.

Para os experimentos com a aplicação BERT foram utilizadas 50 execuções para coletar os dados da versão original, assumida como sem defeito, também chamados de dados de treino, pois estes são utilizados para criar o perfil de referência (ou de uso base) pela metodologia Tricorder. Sendo feito o agrupamento dos dados de treino, foram então adicionados os dados monitorados durante 30 execuções de cada carga de trabalho das versões com defeitos, também chamados de dados de teste.

Conforme apresentado na Tabela 15 os resultados dos experimentos com a aplicação BERT mostram a detecção total das versões com defeito e nenhuma detecção das versões *CONTROL*. Estes resultados foram obtidos sem que fossem necessários ajustes de parâmetros ou verificações de condições, demonstrando que a Tricorder foi eficaz em detectar anomalias nas versões com defeitos sem a necessidade de investimento de tempo para ajustes e calibrações.

O software BERT possui um perfil de desempenho onde as cargas de trabalho apresentam usos de recursos distintos entre si, com similaridades entre algumas das cargas, como as duas versões de SQuAD, mas de forma geral um perfil uniforme e com poucas sobreposições entre suas cargas de trabalho, conforme caracterizado no Capítulo 5. Com base nisso, estes resultados demonstram que a metodologia Tricorder pode ser eficaz na detecção de comportamentos anômalos em aplicações com perfil similar ao desta aplicação.

6.3 *Chess-Alpha Zero* - Resultados e Discussão

No caso do software *Chess-Alpha Zero*, as versões com falhas são capazes de utilizar todas as cargas de trabalho, assim na tabela de resultados são apresentados somente "OK", conforme explicação anterior, números, indicando a quantidade de dados de execuções que foram adicionadas ao agrupamento até que a metodologia Tricorder indicasse a existência de uma anomalia e porcentagens, indicando a taxa de acerto da metodologia. Os resultados obtidos nos experimentos com a aplicação *Chess-Alpha Zero* são apresentados na Tabela 16.

Os resultados com esta aplicação também mostram a eficácia da metodologia Tricorder,

Tabela 16 – Resultados *Chess-Alpha Zero*

CARGAS	API	*CONC*	*LOGIC*	*MEMORY*	*MODEL*	*PROCESS*	*TRAIN*	*CONTROL*	ACERTO
OPT	16	14	14	9	19	16	5	OK	100,0%
OPT_STEP	12	16	7	6	5	5	6	OK	100,0%
EVAL	14	15	20	12	5	8	11	OK	100,0%
SELF	9	15	24	19	4	13	5	OK	100,0%
SUP_LEARN	15	7	26	21	10	22	24	OK	100,0%
ACERTO	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%

Fonte: Elaborada pelo autor.

pois a mesma detectou 100% das anomalias em todas as cargas das versões com defeito, e classificou corretamente 100% das cargas da versão *CONTROL* como sem anomalias. Assim como nos experimentos com BERT, para a aplicação *Chess-Alpha Zero* não foram necessários ajustes, sendo utilizada a metodologia Tricorder nos moldes apresentados anteriormente neste trabalho, com a única alteração de que foram utilizadas 50 execuções de treino ao invés de 30, como nos experimentos realizados por Montes (MONTES, 2019).

Os bons resultados neste experimento podem ser atribuídos às mesmas características encontradas no projeto BERT, onde as cargas de trabalho possuem perfis distintos, com comportamentos consistentes entre execuções, de forma que o perfil criado com os dados de treino seja uniforme a ponto da metodologia ser capaz de detectar até mesmo anomalias com pequenos impactos de desempenho, como é o caso do *LOGIC*, que apresenta uma pequena mudança no fluxo de execução, conforme apresentado no Capítulo 5.

Apesar de apresentar características similares à aplicação anterior, as cargas de trabalho do software *Chess-Alpha Zero* apresentam um uso de recursos significativamente diferente do uso de recursos das cargas de BERT. Isso demonstra que a metodologia Tricorder foi capaz de alcançar altas taxas de detecção de anomalias quando aplicada em sistemas de software com perfis de desempenho consistentes e uniformes, mesmo em aplicações com perfis distintos.

6.4 *Real-Time Voice Cloning* - Resultados e Discussão

Diferentemente das duas aplicações anteriores, os resultados obtidos nos primeiros experimentos com o software *Real-Time Voice Cloning* apresentaram alguns falsos negativos, no caso de versões com defeitos que não foram detectadas, e falsos positivos, no caso de execuções *CONTROL* que foram detectadas como anomalia, conforme apresentado na Tabela 17. Nesta tabela, as células com uma marcação "X", em vermelho, representam defeitos não identificados como anomalia. Na coluna *CONTROL*, referente a versão que não sofreu injeção de defeitos, números em vermelho representam o número de execuções em que a metodologia Tricorder detectou, erroneamente, uma anomalia.

Tabela 17 – Resultados Iniciais *Real-Time Voice Cloning*

CARGAS	API	*CONC*	*LOGIC*	*MEMORY*	*MODEL*	*PROCESS*	*TRAIN*	*CONTROL*	ACERTO
MIN_LONG	28	25	18	12	30	8	21	OK	100,0%
MIN_MEDIUM	X	14	20	X	X	10	28	17	50,0%
MIN_SHORT	30	12	28	28	29	22	X	OK	87,5%
SEC_LONG	28	16	19	21	19	24	X	19	75,0%
SEC_MEDIUM	X	X	26	10	25	X	21	29	50,0%
SEC_SHORT	11	9	23	16	17	24	X	23	75,0%
ACERTO	66,7%	83,3%	100,0%	83,3%	83,3%	83,3%	50,0%	33,3%	72,9%

Fonte: Elaborada pelo autor.

Devido a estes problemas, foi realizada uma investigação, a fim de verificar o comportamento da Tricorder frente a diferentes configurações da metodologia. Realizando experimentos com 30, 40, 50 e 60 execuções de treino, pode-se observar um padrão de aumento de falsos negativos e diminuição de falsos positivos conforme o número de execuções de treino era aumentada. Estes resultados estão sumarizados na Tabela 18.

Tabela 18 – Comparação de Experimentos *Real-Time Voice Cloning*

Execuções Treino	**Métricas Removidas**	**Porcentagem Detecção**	**Porcentagem *CONTROL***
30	N/A	72,9%	33,3%
40	N/A	56,2%	33,3%
50	N/A	50,0%	50,0%
60	N/A	50,0%	66,7%
30	Acessos de I/O	89,5%	50,0%
40	Acessos de I/O	77,0%	50,0%
50	Acessos de I/O	66,7%	50,0%
30	Bytes de I/O	91,6%	50,0%
40	Bytes de I/O	70,8%	50,0%
50	Bytes de I/O	68,7%	50,0%
30	VRAM	83,3%	33,3%
40	VRAM	60,4%	50,0%
50	VRAM	39,5%	50,0%
30	RAM	91,6%	0,0%
40	RAM	89,5%	83,3%
50	**RAM**	**89,5%**	**100,0%**

Mediante os resultados com a alteração de execuções de treino, foram realizados experimentos com a remoção de dados de uma das seis métricas definidas, conforme apresentado na Tabela 18. Primeiramente foram removidas individualmente as métricas de Acessos de I/O e Bytes de I/O, por apresentarem uma variação relativamente pequena dentre as cargas do tipo *SEC* e do tipo *MIN*, conforme apresentado no Capítulo 5.

Conforme apresentado na Tabela 18, com estes experimentos ainda não foi possível obter resultados 100% corretos para a versão *CONTROL*. Por este motivo foram realizados novos

experimentos removendo as métricas RAM e VRAM, que mediante análise realizada utilizando a biblioteca Pandas do Python e o método de Pearson (MCKINNEY, 2021), apresentaram a maior correlação dentre todos os pares de métricas, assim indicando que a remoção de uma das duas métricas poderia ser benéfica, mantendo somente métricas com comportamentos distintos.

A análise de correlação é apresentada na Figura 21. Nesta figura são apresentadas todas as métricas nos eixos X e Y, e em cada uma das células é apresentado um valor de -1 a 1, onde 0 indica 0% de correção entre as métricas indicadas nos dois eixos e, linearmente, -1 ou 1 indicam 100% de correlação.

Figura 21 – Análise de Correlação *Real-Time Voice Cloning*

Fonte: Elaborada pelo autor.

Finalmente, a partir destes experimentos com 50 execuções de treino e removendo os dados da métrica RAM, foram obtidos melhores resultados, conforme a linha destacada em negrito da Tabela 18 e conforme apresentado na Tabela 19. Os resultados da Tabela 19, com esta configuração, mostram que a metodologia Tricorder foi capaz de detectar corretamente 89,5% dos itens, identificando corretamente 100% das cargas de *CONTROL*, e identificando corretamente a maior parte das execuções com defeito.

Comparando a caracterização das cargas de trabalho desta aplicação com as de BERT e *Chess-Alpha Zero*, é possível verificar que esta aplicação apresenta múltiplas cargas de trabalho com comportamentos similares, enquanto as outras aplicações apresentam comportamentos

Tabela 19 – Resultados *Real-Time Voice Cloning*

CARGAS	API	*CONC*	*LOGIC*	*MEMORY*	*MODEL*	*PROCESS*	*TRAIN*	*CONTROL*	ACERTO
MIN_LONG	29	21	8	21	15	16	7	OK	100,0%
MIN_MEDIUM	16	17	20	20	13	X	9	OK	87,5%
MIN_SHORT	6	X	24	17	16	27	X	OK	75,0%
SEC_LONG	14	19	27	10	14	X	17	OK	87,5%
SEC_MEDIUM	9	30	10	8	22	20	14	OK	100,0%
SEC_SHORT	X	15	29	18	29	23	17	OK	87,5%
ACERTO	83,3%	83,3%	100,0%	100,0%	100,0%	66,7%	83,3%	100,0%	89,5%

Fonte: Elaborada pelo autor.

distintos ou com pouca similaridade entre as cargas executadas. A presença de cargas de trabalho com comportamentos muito similares (com uma maior interseção no uso dos recursos) impactou o perfil de desempenho criado pela DAMICORE neste caso, de forma que a metodologia Tricorder teve maiores dificuldades em detectar comportamentos anômalos.

6.5 *Decision-Tree* - Resultados e Discussão

Similarmente aos experimentos com o *Real-Time Voice Cloning*, os resultados iniciais obtidos com a aplicação *Decision-Tree*, utilizando 50 execuções de treino, apresentaram falsos negativos nas execuções das versões com defeito, e falsos positivos em algumas das execuções da versão *CONTROL* conforme ilustra a Tabela 20. Por este motivo, foi realizada uma investigação para identificar a causa destes resultados e buscar mudanças de parâmetros com o objetivo analisar o comportamento da Tricorder.

Tabela 20 – Resultados Iniciais *Decision-Tree*

CARGAS	API	*CONC*	*LOGIC*	*MEMORY*	*MODEL*	*PROCESS*	*TRAIN*	*CONTROL*	ACERTO
OIL_MAMMO	10	6	27	X	X	8	X	OK	62,5%
CANCER_PHON	X	13	11	X	X	6	X	OK	50,0%
IONO_SOLAR	X	26	X	X	X	10	X	OK	37,5%
BANK_PIMA	X	12	X	X	X	17	X	OK	37,5%
WINE_HEART	24	13	X	X	X	4	X	OK	50,0%
IRIS_HABER	X	21	X	X	X	5	23	10	37,5%
ACERTO	33,3%	100,0%	33,3%	0,0%	0,0%	100,0%	16,7%	83,3%	45,8%

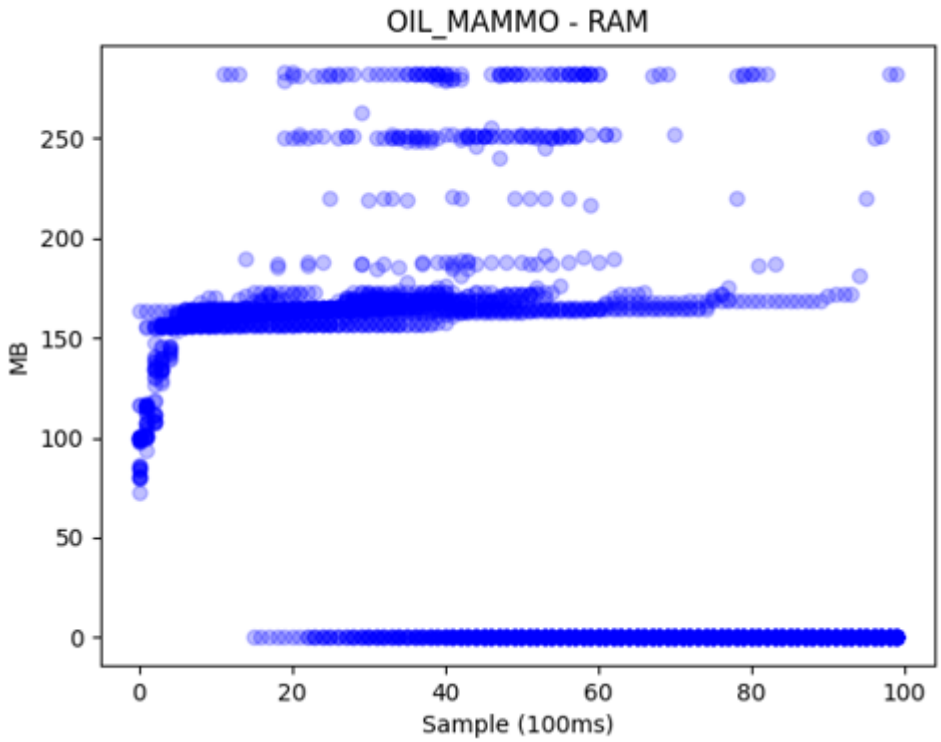
Fonte: Elaborada pelo autor.

Analisando o código e o comportamento desta aplicação, foi possível detectar que ela apresenta fatores que dificultam a detecção pela metodologia Tricorder: execuções curtas; alta variação de uso de recursos entre execuções de uma mesma carga; baixo impacto de defeitos, e; uso de recursos similares entre cargas. Estes fatores fazem com que o perfil de desempenho criado para esta aplicação tenha pouca definição, dificultando a detecção de anomalias pela metodologia.

Investigando o código da aplicação, foi possível verificar que durante a sua execução, a aplicação realiza download de nove das doze bases de dados utilizadas para o processamento. Estas chamadas de I/O são realizadas antes do início do processamento e fazem com que a aplicação espere pela resposta do servidor, o que está sujeito a atrasos. Assim, em cada execução, o início do processamento ocorre em um momento diferente, de forma que o perfil de desempenho criado contemple uma grande variedade de situações.

Na Figura 22 são apresentados todos os pontos coletados da aplicação ao decorrer de 30 execuções usadas para monitoramento. No eixo X da figura são apresentadas as amostras coletadas a cada 100ms e no eixo Y os valores de cada amostra. Analisando esta figura, é possível observar a grande variedade de valores coletados em cada uma das execuções.

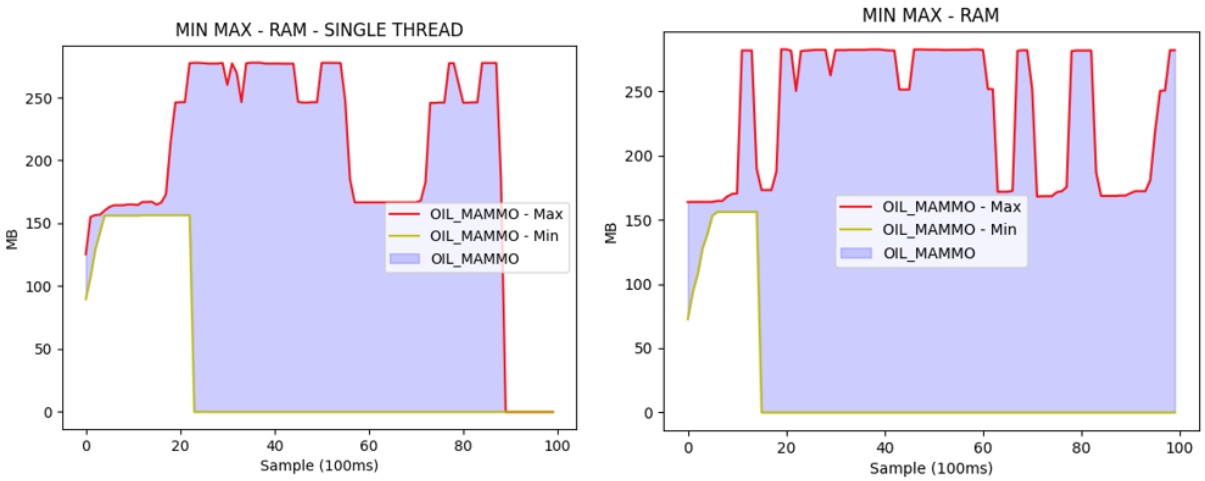
Figura 22 – Amostras *Decision-Tree*



Fonte: Elaborada pelo autor.

O grande impacto de I/O na variação de comportamentos da aplicação *Decision-Tree* é acentuado por uma variação característica do projeto de Aprendizagem de Máquina e também pelo uso de múltiplos *threads*. Nas Figuras 23a e 23b são apresentados os gráficos de valor máximo e mínimo para cada um dos instantes coletados ao decorrer das 30 execuções. Nestes gráficos é possível verificar que mesmo sem o uso de múltiplos *threads* a aplicação apresenta uma grande variação entre valores máximos e mínimos, e com a implementação do uso da biblioteca *multithreading* do Python, essa variação é acentuada.

Além da grande variação do projeto *Decision-Tree*, esta aplicação apresenta rápido processamento e uso de bases de dados pequenas, de forma que os defeitos injetados geram



(a) *Single-Thread* (b) *Multi-Thread*

Figura 23 – *Decision-Tree* - Valores máximos e mínimos

Fonte: Elaborada pelo autor.

impactos pequenos em relação à variação entre execuções. Isto faz com que a metodologia tenha dificuldades em detectar os defeitos como anomalias, pois os impactos causados podem ser enquadrados dentro do perfil de desempenho base.

O terceiro fator identificado é a similaridade entre as cargas de trabalho. Analisando os dados de agrupamento, pode-se observar que as características desta aplicação fazem com que ocorram agrupamentos entre dados de diferentes cargas de trabalho, inclusive com dados de versões com falhas, fazendo com que os dados sejam classificados em grupos de outras cargas ao invés de criar novos grupos exclusivos, o que geraria uma anomalia.

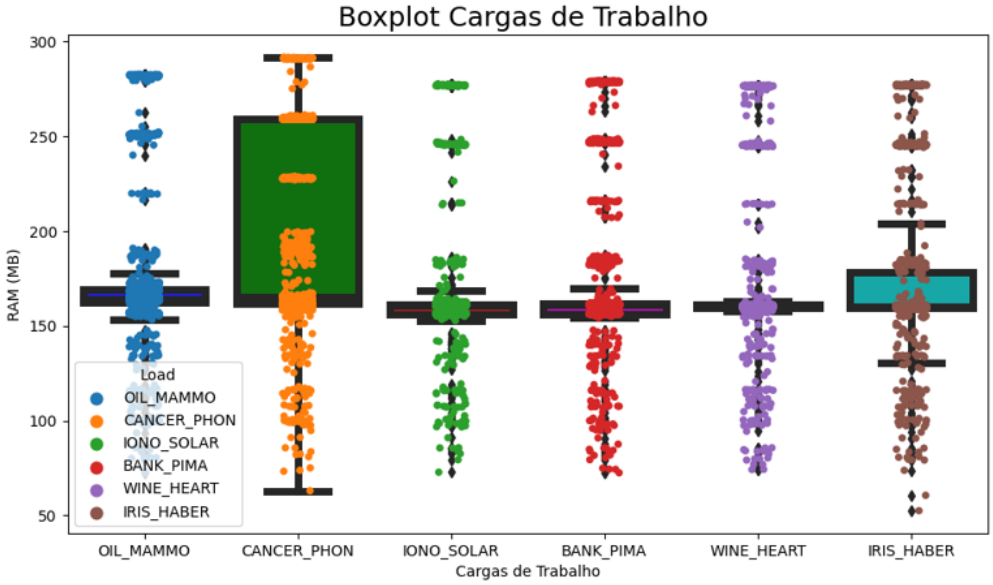
Na Figura 24 são apresentados os *box plots* de cada uma das cargas de trabalho da aplicação *Decision-Tree*. Nela é possível verificar a grande dispersão de valores de uso de recurso e também a similaridade de uso de recursos entre todas as cargas de trabalho.

A sobreposição entre as cargas de trabalho pode ser observada também na Figura 25. Nesta figura são exibidos os valores de mínimo e máximo de cada uma das cargas de trabalho, sendo possível verificar que estas cargas apresentam perfis próximos.

Considerando as características deste software e os resultados obtidos, duas modificações foram realizadas durante a execução dos experimentos: aumento do tempo de execução de 6s, como foi proposto originalmente, para 10s, e; redução do número de execuções de treino, de forma a minimizar a dispersão da aplicação. Foram realizados também experimentos com a remoção de métricas (as métricas X e Y foram removidas dos arquivos de rastro), mas estes, diferentemente do esperado, não trouxeram resultados melhores, como ocorreu nos experimentos com o software *Real-Time Voice Cloning*.

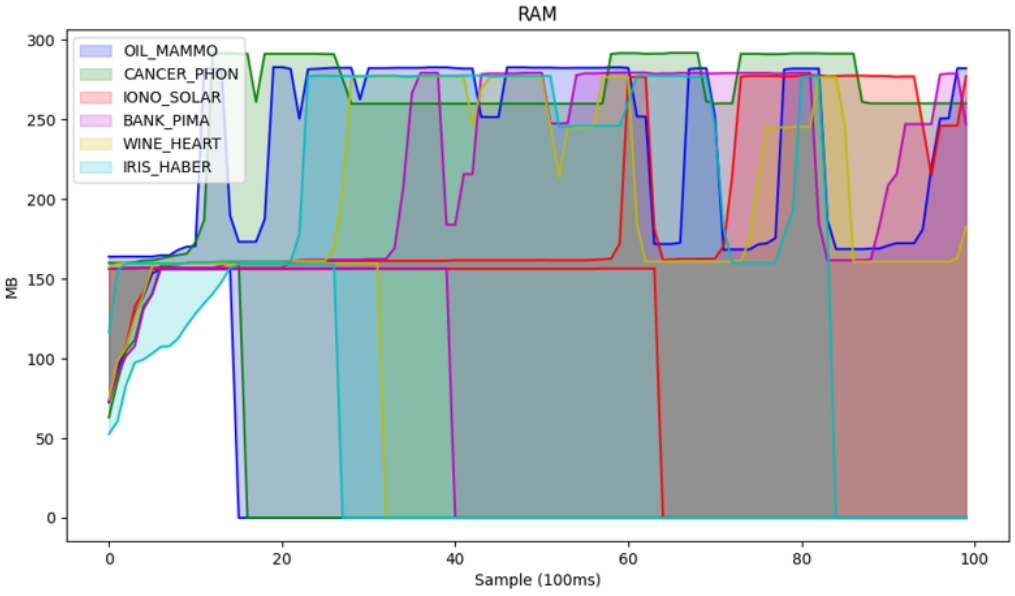
Com estas duas modificações, especialmente a redução de execuções, foi possível verificar que os resultados obtidos foram melhorados, por exemplo, experimentos com 60 execuções

Figura 24 – *Box plots Decision-Tree*



Fonte: Elaborada pelo autor.

Figura 25 – Sobreposição *Decision-Tree*



Fonte: Elaborada pelo autor.

de treino apresentavam uma taxa de acerto de cerca de 54%, conforme apresentado na Tabela 21, enquanto experimentos com 30 execuções de treino apresentaram cerca de 79% de acerto, conforme apresentando na Tabela 22.

Tabela 21 – Resultados com 60 Execuções *Decision-Tree*

CARGAS	API	*CONC*	*LOGIC*	*MEMORY*	*MODEL*	*PROCESS*	*TRAIN*	*CONTROL*	ACERTO
OIL_MAMMO	X	14	20	X	X	12	22	OK	62,5%
CANCER_PHON	X	28	X	X	X	11	X	OK	37,5%
IONO_SOLAR	24	21	27	X	X	14	5	OK	75,0%
BANK_PIMA	X	X	X	X	X	6	X	OK	25,0%
WINE_HEART	X	23	25	X	X	11	X	OK	50,0%
IRIS_HABER	X	12	X	6	18	11	20	OK	75,0%
ACERTO	16,7%	83,3%	50,0%	16,7%	16,7%	100,0%	50,0%	100,0%	54,2%

Fonte: Elaborada pelo autor.

Tabela 22 – Resultados com 30 Execuções *Decision-Tree*

CARGAS	API	*CONC*	*LOGIC*	*MEMORY*	*MODEL*	*PROCESS*	*TRAIN*	*CONTROL*	ACERTO
OIL_MAMMO	X	24	25	23	24	6	26	OK	87,5%
CANCER_PHON	X	20	X	26	X	20	15	15	50,0%
IONO_SOLAR	X	14	23	12	20	15	18	OK	87,5%
BANK_PIMA	20	8	X	X	30	16	22	OK	75,0%
WINE_HEART	11	28	11	30	25	9	23	OK	100,0%
IRIS_HABER	X	22	22	X	5	8	13	OK	75,0%
ACERTO	33,3%	100,0%	66,7%	66,7%	83,3%	100,0%	100,0%	83,3%	79,2%

Fonte: Elaborada pelo autor.

Apesar dos resultados obtidos serem melhores, foi observado que devido à alta variabilidade do uso de recursos pela aplicação *Decision-Tree*, os monitoramentos apresentam dados diferentes, de forma que os resultados variam significativamente entre experimentos. Assim, foi optado por realizar a execução deste experimento múltiplas vezes, a fim de verificar se a metodologia apresentaria bons resultados, mesmo com diferentes variações, e extrair as médias dos resultados. Na Tabela 23 são apresentados os resultados obtidos com cinco experimentos por célula carga/defeito, e calculando a quantidade de acertos pela metodologia. Na tabela, a coluna e linha MÉDIA representam a média da taxa de acertos de cada uma das cargas de trabalho e defeitos, respectivamente. A célula do canto inferior direito, destacada em uma cor diferente, apresenta a média total de acertos, considerando todas as células.

Analisando a média de acertos nos cinco experimentos, é possível justificá-las com base nas características das cargas e defeitos, descritas no Capítulo 5. A Tricorder apresentou

Tabela 23 – Estatísticas *Decision-Tree*

CARGAS	API	*CONC*	*LOGIC*	*MEMORY*	*MODEL*	*PROCESS*	*TRAIN*	*CONTROL*	MÉDIA
OIL_MAMMO	60,0%	20,0%	60,0%	40,0%	40,0%	100,0%	100,0%	100,0%	65,0%
CANCER_PHON	40,0%	100,0%	0,0%	60,0%	40,0%	100,0%	100,0%	80,0%	65,0%
IONO_SOLAR	60,0%	80,0%	60,0%	80,0%	40,0%	100,0%	100,0%	80,0%	75,0%
BANK_PIMA	60,0%	100,0%	20,0%	60,0%	60,0%	100,0%	80,0%	80,0%	70,0%
WINE_HEART	60,0%	80,0%	40,0%	40,0%	20,0%	100,0%	40,0%	80,0%	57,5%
IRIS_HABER	60,0%	100,0%	40,0%	20,0%	40,0%	100,0%	40,0%	80,0%	60,0%
MÉDIA	56,7%	80,0%	36,7%	50,0%	40,0%	100,0%	76,7%	83,3%	65,4%

Fonte: Elaborada pelo autor.

uma taxa de detecção maior para defeitos com altos impactos no desempenho da aplicação e também detectou corretamente a ausência de anomalias na maioria dos experimentos com as cargas da versão *CONTROL*, porém apresentou baixa detecção de defeitos com baixo impacto no desempenho (como esperado).

Apesar das dificuldades destacadas, a metodologia Tricorder ainda apresentou mais de 60% de taxa de detecção de defeitos. Considerando que a metodologia é capaz de detectar defeitos que não afetam diretamente as saídas, de forma que estes são difíceis de se detectar por outras técnicas, este resultado é considerado muito positivo, demonstrando que a Tricorder é promissora mesmo em condições adversas para a mesma.

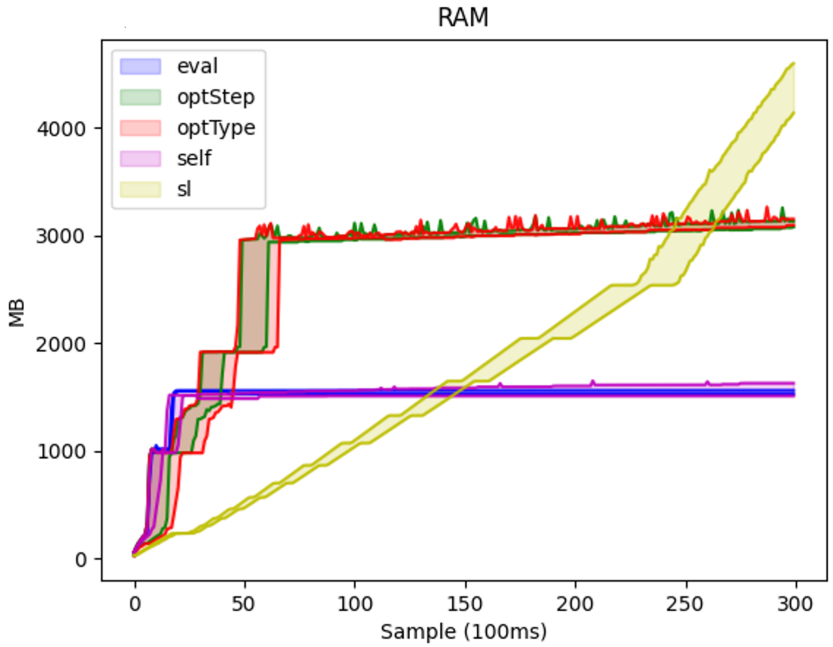
6.6 Comparação de Resultados

Analisando os resultados obtidos nos experimentos realizados é possível verificar que a Tricorder, de uma forma geral, conseguiu revelar uma grande quantidade de defeitos, mesmo em situações que poderiam ser consideradas adversas à metodologia proposta. A metodologia apresentou menos êxito nos experimentos com aplicações com perfis de desempenho menos uniformes, mas foi capaz de detectar corretamente as anomalias, apresentando uma taxa de detecção ainda maior mediante a realização de ajustes.

Comparando os perfis de desempenho das aplicações, conforme apresentado no Capítulo 5, os projetos *Decision-Tree* e *Real-Time Voice Cloning* possuem cargas de trabalho similares entre si, apresentando uma sobreposição no uso de recursos entre suas cargas. Em contraste, as cargas de trabalho das aplicações BERT e *Chess-Alpha Zero* apresentam desempenhos distintos, com pouca similaridade. Estes comportamentos podem ser observados nas Figuras 26 e 27, que apresentam o uso de memória RAM de todas as cargas de trabalho das aplicações *Chess-Alpha Zero* e *Real-Time Voice Cloning*, respectivamente.

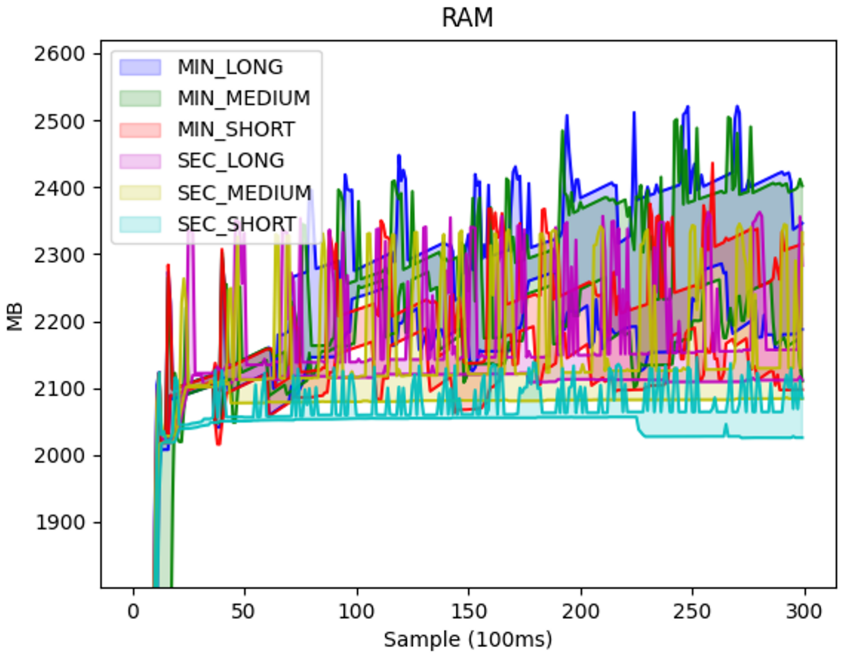
Analisando as figuras, é possível observar que entre todas as cargas de trabalho da *Real-Time Voice Cloning* existe uma sobreposição, indicando que as cargas de trabalho apresentam comportamentos similares, com baixa distinção entre si. Isso faz com que o perfil de desempenho criado pela metodologia Tricorder abranja uma grande variedade de usos de recursos. Desta

Figura 26 – Sobreposição *Chess-Alpha Zero*



Fonte: Elaborada pelo autor.

Figura 27 – Sobreposição Real-Time Voice Cloning



Fonte: Elaborada pelo autor.

forma, caso uma carga de trabalho apresente um uso de memória elevado devido a um defeito, este comportamento anômalo pode ser classificado pela metodologia como pertencente a uma outra carga, ao invés de ser detectado corretamente como uma anomalia.

No caso da aplicação *Decision-Tree*, esta situação adversa é intensificada, principalmente pela variação no tempo, causada pelo I/O, de forma que este software apresente a menor taxa de detecção inicial dentre os quatro projetos selecionados para este estudo. Apesar disso, mesmo com estas dificuldades a metodologia foi capaz de detectar todas as anomalias, em ao menos uma carga de trabalho.

Considerando que a metodologia Tricorder utiliza o perfil de desempenho criado para conseguir detectar anomalias, era esperado que a Tricorder apresentasse maiores dificuldades em aplicações com perfis pouco uniformes e com alta variação. Porém os resultados obtidos com os experimentos propostos foram melhores do que esperado mesmo na aplicação com a maior quantidade de condições adversas, *Decision-Tree*. Nela, a metodologia foi capaz de classificar corretamente a maioria das cargas de trabalho, detectar a não existência de defeitos em quase todas os experimentos com *CONTROL* e detectar todos os defeitos.

6.7 Considerações Finais

Os resultados obtidos nos experimentos com a metodologia Tricorder se demonstraram promissores, com resultados positivos na detecção de defeitos das aplicações utilizadas. Apesar disso, foi possível observar características de software que impactam negativamente a eficiência da metodologia, sendo necessária a realização de novos estudos a fim de aumentar a taxa de detecção de defeitos nas aplicações.

As cargas definidas para cada uma das aplicações têm como objetivo definir um perfil de uso que seja abrangente e contemple o desempenho normal de cada uma das aplicações, de forma que um comportamento fora deste perfil possa ser considerado como anormal. No caso das aplicações BERT e *Chess-Alpha Zero*, as cargas apresentam padrões uniformes, o que auxiliam a criação de um perfil mais definido e com maior facilidade para detecção de comportamentos anômalos. No caso das aplicações *Real-Time Voice Cloning* e *Decision-Tree*, as cargas de trabalho apresentam comportamentos similares, de forma que tornou-se difícil a detecção de anomalias quando as versões com falhas apresentam comportamentos que possam ser enquadrados em outras cargas de trabalho.

Apesar das dificuldades, os resultados obtidos foram melhores do que esperado. A metodologia foi capaz de classificar corretamente todas as cargas das aplicações BERT e *Chess-Alpha Zero*, e detectou a maioria dos defeitos das aplicações *Real-Time Voice Cloning* e *Decision-Tree*, que apresentam condições adversas.

CAPÍTULO

7

CONCLUSÃO

7.1 Análise dos Resultados

Este trabalho foi realizado com o objetivo de verificar a eficácia de metodologia Tricorder quando empregada na revelação de defeitos em aplicações de Aprendizagem de Máquina. Este estudo seguiu a hipótese de que a metodologia Tricorder pode ser utilizada como uma etapa complementar de teste dessas aplicações, sendo usada na fase de manutenção ou durante o fluxo de MLOps. A Tricorder foi capaz de detectar defeitos nos experimentos que não seriam facilmente identificados por um oráculo de teste, quando fossem usadas outras técnicas tradicionais de teste, pelo fato desses defeitos não interromperem a execução da aplicação nem produzirem resultados fora da faixa normal de resultados esperados (i.e., falhas).

Os experimentos realizados obtiveram sucesso na detecção de defeitos, conforme apresentado no Capítulo 6. Nas aplicações *Real-Time Voice Cloning* e *Decision-Tree* que apresentam grande variação e pouca definição, foi necessário investimento de tempo para que a metodologia fosse calibrada, mas a partir deste investimento inicial a Tricorder foi capaz de detectar corretamente um maior número de anomalias, apresentando taxa de acerto de 89,5% para o software *Real-Time Voice Cloning* e uma média de acertos de 65,4% para o software *Decision-Tree*. Mesmo sem a calibração citada, a metodologia foi capaz de detectar 72,9% e 45,8% dos defeitos inseridos nestes *benchmarks*, respectivamente.

Quando aplicada nos programas BERT e *Chess-Alpha Zero* que apresentam baixa variação no consumo de recursos computacionais e perfil de desempenho definido, a metodologia Tricorder detectou 100% das anomalias inseridas, em todas as versões das aplicações de Aprendizagem de Máquina com falhas. Nestes cenários os resultados foram obtidos sem que fosse necessário investimento de tempo para calibrar a aplicação.

Estes resultados acima demonstram que, conforme a hipótese seguida neste trabalho, a Tricorder teve eficácia na revelação de defeitos nos experimentos realizados, especialmente

considerando que a metodologia é proposta para ser usada como instrumento complementar de teste, sendo capaz de detectar defeitos que não afetam as saídas esperadas para as aplicações, e podendo ser aplicada com baixo investimento.

7.2 Contribuições

Conforme os objetivos destacados anteriormente, o desenvolvimento deste estudo contribui para o estado da arte no que diz respeito à VV&T de aplicações de Aprendizagem de Máquina. A metodologia Tricorder demonstrou-se eficaz neste contexto, permitindo a execução de testes com baixo investimento de tempo e esforço em alguns contextos, por não necessitar da criação de casos de teste e acompanhamento constante de um testador.

Especificamente, os resultados deste trabalho contribuem para a garantia de qualidade de software, pois afere uma nova abordagem de teste que pode ser usada como complemento aos testes de regressão durante a fase de manutenção de software, ou até mesmo durante um fluxo de MLOps, sendo capaz de detectar defeitos desconhecidos.

Essas contribuições são principalmente direcionadas para o desenvolvimento de aplicações de Aprendizagem de Máquina, as quais possuem um número limitado de abordagens de teste devido à sua complexidade e uso de ferramentas e bibliotecas especializadas. Como a metodologia Tricorder executa os testes de forma caixa-preta, sem que seja necessário acesso ao código fonte, ela pode contribuir significativamente para a garantia de qualidade desse tipo de aplicação.

Os resultados apresentados neste trabalho podem também ser utilizados em novas pesquisas, contribuindo para a área de VV&T no geral e especialmente para Aprendizagem de Máquina. Finalmente, este estudo contribui com o desenvolvimento da metodologia Tricorder, promovendo o seu uso pela comunidade de testes de software e Aprendizagem de Máquina de forma que ela possa ser constantemente aperfeiçoada por meio de novos estudos e desenvolvimentos de pesquisa.

7.3 Limitações

No desenvolvimento deste trabalho foi possível verificar que o desempenho da metodologia Tricorder foi impactado pela baixa definição das cargas de trabalho de uma das aplicações selecionadas. Porém, apesar de poder ser concluído que as cargas de trabalho têm uma grande importância para a metodologia, não foi investigada com mais profundidade a eficácia da metodologia quando utilizadas diferentes cargas de trabalho, ou diferentes combinações destas.

Devido a flexibilidade da Tricorder, é possível que o uso de outras configurações e, especificamente, de outras cargas de trabalho tragam resultados melhores para aplicações que

apresentam limitações, como é o caso da *Decision-Tree*. Neste trabalho não foram exploradas estas opções, com o intuito de obter melhores resultados neste contexto.

7.4 Trabalhos Futuros

Como discutido no Capítulo 6 a metodologia se demonstrou eficaz nos experimentos, sendo capaz de classificar corretamente a maioria das cargas de trabalho e alcançando uma taxa de acerto de 100% em duas das quatro aplicações. Porém em duas destas aplicações foi necessário um investimento de tempo para calibrar a metodologia. Considerando que a calibração foi basicamente realizada por meio de experimentos com parâmetros diferentes, é possível que esta etapa de calibração seja automatizada, de forma que não seja necessário um grande acompanhamento humano, e permitindo que as melhores condições sejam utilizadas nos testes reais.

Considerando também o levantamento feito sobre os monitores, apresentado na sessão Avaliação de Monitores do Capítulo 5, sugere-se um estudo futuro mais aprofundado sobre este tópico, com o objetivo de encapsular todos estes monitores, blindando a Tricorder de interferir diretamente em seus códigos, mas permitindo que a metodologia possa utilizá-los sem maiores esforços.

Como apresentado na sessão Limitações, seria interessante também a realização de um estudo detalhado sobre o impacto do uso de diferentes cargas de trabalho na metodologia Tricorder. Com este trabalho podemos concluir que estas têm grande importância para a metodologia, mas é importante investigar especificamente quais condições, quantidades, variações e outros parâmetros que possam trazer melhores resultados em aplicações com diferentes comportamentos.

Por fim, é interessante que seja estudada com maior profundidade a forma que a Tricorder utiliza a metodologia DAMICORE. Nos experimentos realizados até o momento, foi utilizada a mesma abordagem de interpretação, na qual a Tricorder considera como uma anomalia a criação de um novo grupo apenas com dados de uma nova versão. Porém, é possível que existam formas mais eficientes de análise dos agrupamentos, por exemplo, verificando as características dos grupos até que um limiar seja alcançado, ao invés de aguardar a criação de um novo grupo somente com os dados da nova versão.

REFERÊNCIAS

ALAM, K. A.; AHMAD, R.; KO, K. Enabling far-edge analytics: Performance profiling of frequent pattern mining algorithms. **IEEE Access**, v. 5, p. 8236–8249, 2017. Citado na página 53.

ALPAYDIN, E. **Introduction to Machine Learning**. [S.l.]: The MIT Press, 2014. ISBN 0262028182. Citado nas páginas 21, 30 e 31.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. [S.l.]: Cambridge University Press, 2008. ISBN 0521880386. Citado nas páginas 40 e 44.

ARORA, N.; BELL, J.; IVANI, F.; KAISER, G.; RAY, B. Replay without recording of production bugs for service oriented applications. In: . [S.l.]: Association for Computing Machinery, 2018. (ASE 2018), p. 452–463. ISBN 9781450359375. Citado na página 53.

AWAD, M.; KHANNA, R. **Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers**. 1st. ed. USA: Apress, 2015. ISBN 1430259892. Citado nas páginas 30, 31, 32 e 33.

BANERJEE, A.; SRIVASTAVA, A. A cloud performance analytics framework to support online performance diagnosis and monitoring tools. In: . [S.l.]: Association for Computing Machinery, 2019. (ICPE '19), p. 151–158. ISBN 9781450362399. Citado na página 53.

BAUM, L. F. **Wizard of Oz**. 1900. Disponível em: <http://read.gov/books/oz.html>. Citado na página 90.

____. **Wizard of Oz Audio**. 2021. Disponível em: <https://etc.usf.edu/lit2go/158/the-wonderful-wizard-of-oz/>. Citado na página 90.

BELKHIRI, A.; DAGENAIS, M. Diagnostic and troubleshooting of openflow-enabled switches using kernel and userspace traces. **International Journal of Communication Systems**, v. 34, n. 14, p. e4920, 2021. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/dac.4920>. Citado na página 53.

BELLMAN, R. **An Introduction to Artificial Intelligence: Can Computers Think?** [S.l.]: Boyd & Fraser Publishing Company, 1978. ISBN 9780878350667. Citado na página 28.

BERKELEY, U. of C. **Ganglia Monitoring System**. 2021. Disponível em: <http://ganglia.info/>. Citado nas páginas 48, 49 e 70.

BHATTACHARYYA, A.; AMZA, C. Pret: A tool for automatic phase-based regression testing. In: **2018 IEEE CloudCom**. [S.l.: s.n.], 2018. p. 284–289. ISSN 2330-2186. Citado nas páginas 53 e 56.

BOEHM, B. **Software engineering: R & D trends and defense needs. Research Directions in Software Technology**. [S.l.]: MIT Press Cambridge, 1979. Citado na página 37.

BRAGA, D. **Tricorder Machine Learning**. 2021. Disponível em: <https://github.com/diegobraga92/TricorderMachineLearning>. Citado na página 82.

BRAIN, G. **TensorFlow**. 2021. Disponível em: <https://www.tensorflow.org/>. Citado na página 66.

CASOLA, V.; BENEDICTIS, A. D.; RAK, M.; VILLANO, U. An automatic tool for benchmark testing of cloud applications. In: **CLOSER**. [S.l.: s.n.], 2017. Citado na página 53.

CENTREON. **Centreon**. 2021. Disponível em: <https://www.centreon.com/>. Citado na página 72.

CERVANTES, A. Exploring the use of a test automation framework. In: **2009 IEEE Aerospace conference**. [S.l.: s.n.], 2009. p. 1–9. Citado na página 44.

CESAR, B. K. M. **Estudo e extensão da metodologia DAMICORE para tarefas de classificação**. Dissertação (Mestrado) — Universidade de São Paulo, 2016. Citado nas páginas 59 e 75.

CHANDRASHEKHAR, B. N.; SANJAY, H. A. Performance analysis of sequential and parallel programming paradigms on cpu-gpus cluster. In: **2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)**. [S.l.: s.n.], 2021. p. 1205–1213. Citado na página 53.

CHARNIAK, E.; MCDERMOTT, D. **Introduction to Artificial Intelligence**. [S.l.: s.n.], 1986. ISBN 978-0-201-11945-9. Citado na página 28.

CHEN, C.; WENG, Q.; WANG, W.; LI, B.; LI, B. Semi-dynamic load balancing: efficient distributed learning in non-dedicated environments. In: . [S.l.: s.n.], 2020. p. 431–446. Citado na página 68.

CHEN, H.; MA, H.; CHU, X.; XUE, D. Anomaly detection and critical attributes identification for products with multiple operating conditions based on isolation forest. **Advanced Engineering Informatics**, v. 46, p. 101139, 2020. ISSN 1474-0346. Citado nas páginas 53, 54 e 56.

CHEN, Z.; YAO, H.; LOU, Y.; CAO, Y.; LIU, Y.; WANG, H.; LIU, X. An empirical study on deployment faults of deep learning based mobile applications. In: **2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2021. p. 674–685. Citado nas páginas 79 e 80.

CHIEN, S.; PODOBAS, A.; PENG, I.; MARKIDIS, S. tf-darshan: Understanding fine-grained i/o performance in machine learning workloads. 08 2020. Citado na página 68.

CHOLLET, F. **Keras**. 2021. Disponível em: <https://keras.io/>. Citado na página 66.

CHOWDHURY, F.; ZHU, Y.; HEER, T.; PAREDES, S.; MOODY, A.; GOLDSTONE, R.; MOHROR, K.; YU, W. I/o characterization and performance evaluation of beegfs for deep learning. In: . [S.l.: s.n.], 2019. p. 1–10. ISBN 978-1-4503-6295-5. Citado na página 68.

COLEMAN, C.; KANG, D.; NARAYANAN, D.; NARDI, L.; ZHAO, T.; ZHANG, J.; BAILIS, P.; OLUKOTUN, K.; Ré, C.; ZAHARIA, M. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. **SIGOPS Oper. Syst. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 53, n. 1, p. 14–25, jul. 2019. ISSN 0163-5980. Disponível em: <https://doi.org/10.1145/3352020.3352024>. Citado na página 68.

COURNAPEAU, D. **scikit-learn**. 2021. Disponível em: <https://scikit-learn.org/stable/>. Citado na página 66.

DAI, T.; DEAN, D.; WANG, P.; GU, X.; LU, S. Hytrace: A hybrid approach to performance bug diagnosis in production cloud infrastructures. **IEEE Transactions on Parallel and Distributed Systems**, v. 30, n. 1, p. 107–118, 2019. Citado na página 53.

DELAMARO, M.; JINO, M.; MALDONADO, J. **Introdução ao Teste de Software**. [S.l.]: Elsevier Brasil, 2016. ISBN 9788535283532. Citado nas páginas 36, 37, 38, 39, 40, 41, 42 e 43.

DELGADO-PéREZ, P.; SáNCHEZ, A. B.; SEGURA, S.; MEDINA-BULO, I. Performance mutation testing. **Software Testing, Verification and Reliability**, p. e1728, 2020. Citado na página 53.

____. Performance mutation testing. **Software Testing, Verification and Reliability**, v. 31, n. 5, p. e1728, 2021. E1728 stvr.1728. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1728>. Citado na página 53.

DEVLIN, J. **BERT**. 2020. Disponível em: <https://github.com/google-research/bert>. Citado nas páginas 67 e 82.

DEVLIN, J.; CHANG, M.-W.; LEE, K.; TOUTANOVA, K. **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding**. 2019. Citado nas páginas 33 e 67.

DOšILOVIć, F. K.; BRčIć, M.; HLUPIć, N. Explainable artificial intelligence: A survey. In: **2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)**. [S.l.: s.n.], 2018. p. 0210–0215. Citado na página 21.

DUBE, P.; SURA, Z. Impact of system resources on performance of deep neural network. In: . [S.l.: s.n.], 2018. p. 125–127. Citado na página 68.

FAASSE, S.; BUCEK, J.; SCHMIDT, D. Out of band performance monitoring of server workloads: Leveraging restful api to monitor compute resource utilization and performance related metrics for server performance analysis. In: . [S.l.]: Association for Computing Machinery, 2020. (ICPE '20), p. 4–11. ISBN 9781450369916. Citado na página 53.

FACEBOOK. **PyTorch**. 2021. Disponível em: <https://pytorch.org/>. Citado na página 66.

FISHER, R. **Iris Dataset**. 1936. Disponível em: <https://www.kaggle.com/uciml/iris>. Citado na página 67.

FLACH, P. **Machine Learning: The Art and Science of Algorithms That Make Sense of Data**. USA: Cambridge University Press, 2012. ISBN 1107422221. Citado nas páginas 21, 30 e 31.

FOURNIER, Q.; EZZATI-JIVAN, N.; ALOISE, D.; DAGENAIS, M. R. Automatic cause detection of performance problems in web applications. In: **2019 IEEE ISSREW**. [S.l.: s.n.], 2019. p. 398–405. Citado na página 53.

FU, S.; PRIOR, R.; KIM, H. Dmfd: Non-intrusive dependency inference and flow ratio model for performance anomaly detection in multi-tier cloud applications. In: **2019 IEEE 12th CLOUD**. [S.l.: s.n.], 2019. p. 164–173. Citado nas páginas 53, 55 e 56.

GALIN, D. **Software Quality Assurance: From Theory to Implementation**. [S.l.]: Pearson Education Limited, 2004. ISBN 9780201709452. Citado na página 44.

GALSTAD, E. **Nagios**. 2021. Disponível em: <https://www.nagios.org/>. Citado na página 71.

GRECHANIK, M.; LUO, Q.; POSHYVANYK, D.; PORTER, A. Enhancing rules for cloud resource provisioning via learned software performance models. In: . [S.l.]: Association for Computing Machinery, 2016. (ICPE '16), p. 209–214. ISBN 9781450340809. Citado nas páginas 53 e 54.

GU, J.; LIU, H.; ZHOU, Y.; WANG, X. Deepprof: Performance analysis for deep learning applications via mining gpu execution patterns. 07 2017. Citado nas páginas 53, 54 e 56.

____. Deepprof: Performance analysis for deep learning applications via mining gpu execution patterns. 07 2017. Citado na página 68.

HAUGELAND, J. **Artificial Intelligence: The Very Idea**. USA: Massachusetts Institute of Technology, 1985. ISBN 0262081539. Citado na página 28.

HUMBATOVA, N.; JAHANGIROVA, G.; BAVOTA, G.; RICCIO, V.; STOCCO, A.; TONELLA, P. Taxonomy of real faults in deep learning systems. In: **Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2020. (ICSE '20), p. 1110–1121. ISBN 9781450371216. Disponível em: <https://doi.org/10.1145/3377811.3380395>. Citado nas páginas 79, 80 e 81.

IBRAHIM, Y.; WANG, H. Soft errors in dnn accelerators: A comprehensive review. **Microelectronics Reliability**, v. 115, p. 113969, 10 2020. Citado na página 68.

ICINGA. **Icinga**. 2021. Disponível em: <https://icinga.com/>. Citado na página 71.

ISLAM, M. J.; NGUYEN, G.; PAN, R.; RAJAN, H. A comprehensive study on deep learning bug characteristics. In: **Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2019. (ESEC/FSE 2019), p. 510–520. ISBN 9781450355728. Disponível em: <https://doi.org/10.1145/3338906.3338955>. Citado nas páginas 79 e 81.

ISLAM, T.; MANIVANNAN, D. Predicting application failure in cloud: A machine learning approach. In: **2017 IEEE ICCC**. [S.l.: s.n.], 2017. p. 24–31. Citado nas páginas 53, 55 e 56.

ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary. **ISO/IEC/IEEE 24765:2017(E)**, p. 1–541, 2017. Citado na página 37.

JAIN, R. **The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling**. [S.l.]: Wiley, 1991. Citado nas páginas 45, 46, 47, 48 e 49.

JEMINE, C. **Real-Time Voice Cloning**. 2021. Disponível em: <https://github.com/CorentinJ/Real-Time-Voice-Cloning>. Citado nas páginas 67 e 83.

JEON, M.; VENKATARAMAN, S.; PHANISHAYEE, A.; QIAN, J.; XIAO, W.; YANG, F. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. 01 2019. Citado na página 68.

JIA, L.; ZHONG, H.; WANG, X.; HUANG, L.; LU, X. An empirical study on bugs inside tensorflow. In: NAH, Y.; CUI, B.; LEE, S.-W.; YU, J. X.; MOON, Y.-S.; WHANG, S. E. (Ed.). **Database Systems for Advanced Applications**. Cham: Springer International Publishing, 2020. p. 604–620. Citado nas páginas 79 e 80.

____. The symptoms, causes, and repairs of bugs inside a deep learning library. **Journal of Systems and Software**, v. 177, p. 110935, 2021. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121221000327>. Citado nas páginas 79, 80 e 81.

JIMENEZ, I.; WATKINS, N.; SEVILLA, M.; LOFSTEAD, J.; MALTZAHN, C. quiho: Automated performance regression testing using inferred resource utilization profiles. In: . [S.l.]: Association for Computing Machinery, 2018. (ICPE '18), p. 273–284. ISBN 9781450350952. Citado na página 53.

KHATUYA, S.; GANGULY, N.; BASAK, J.; BHARDE, M.; MITRA, B. Adele: Anomaly detection from event log empiricism. In: **IEEE INFOCOM 2018**. [S.l.: s.n.], 2018. p. 2114–2122. Citado nas páginas 53 e 55.

KHUDUR, A. A.; POLOS, R. P. Gathering multi-dimensional data of scalable container-based cluster for performance analysis. In: **2021 Fourth International Conference on Computational Intelligence and Communication Technologies (CCICT)**. [S.l.: s.n.], 2021. p. 139–145. Citado na página 53.

KIM, M.; KIM, Y.; LEE, E. Denchmark: A bug benchmark of deep learning-related software. In: **2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)**. [S.l.: s.n.], 2021. p. 540–544. Citado na página 79.

KIRK, M. **Thoughtful Machine Learning: A Test-Driven Approach**. [S.l.]: O'Reilly Media, 2014. ISBN 9781449374099. Citado nas páginas 22, 34, 35 e 36.

____. **Thoughtful Machine Learning with Python: A Test-Driven Approach**. [S.l.]: O'Reilly Media, 2017. ISBN 9781491924082. Citado na página 33.

KOU, H.; CHEN, P. An ensemble signature-based approach for performance diagnosis in big data platform. In: **2018 IEEE SOSE**. [S.l.: s.n.], 2018. p. 106–115. Citado nas páginas 53 e 56.

KURZWEIL, R. **The Age of Intelligent Machines**. Cambridge, MA, USA: MIT Press, 1990. ISBN 0262111217. Citado na página 28.

LAABER, C.; LEITNER, P. (hlg)opper: Performance history mining and analysis. In: . [S.l.]: Association for Computing Machinery, 2017. (ICPE '17), p. 167–168. ISBN 9781450344043. Citado na página 53.

LAKSHMANAN, V.; ROBINSON, S.; MUNN, M. **Machine learning design patterns**. [S.l.]: O'Reilly Media, 2020. Citado na página 34.

LASDPC. **Distributed Systems and Concurrent Programming Laboratory**. 2021. Disponível em: <http://lasdpc.icmc.usp.br/>. Citado na página 51.

LEVIN, D. **Strace**. 2021. Disponível em: <https://man7.org/linux/man-pages/man1/strace.1.html>. Citado na página 73.

LI, P.; LUO, Y.; ZHANG, N.; CAO, Y. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. In: **2015 IEEE International Conference on Networking, Architecture and Storage (NAS)**. [S.l.: s.n.], 2015. p. 347–348. Citado na página 68.

LI, S.; JIA, Z.; LI, Y.; LIAO, X.; XU, E.; LIU, X.; HE, H.; GAO, L. Detecting performance bottlenecks guided by resource usage. **IEEE Access**, v. 7, p. 117839–117849, 2019. Citado nas páginas 53, 54 e 56.

LIAO, L.; CHEN, J.; LI, H. Using black-box performance models to detect performance regressions under varying workloads: an empirical study. **Empir Software**, 2020. Citado na página 68.

LINUX. **Perf**. 2021. Disponível em: <https://perf.wiki.kernel.org/index.php/Main_Page>. Citado na página 73.

LOMIO, F.; JURVANSUU, S.; TAIBI, D. Metrics selection for load monitoring of service-oriented system. In: **Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution**. New York, NY, USA: Association for Computing Machinery, 2021. (MaLTESQuE 2021), p. 31–36. ISBN 9781450386258. Disponível em: <https://doi.org/10.1145/3472674.3473983>. Citado na página 53.

LUCAS, H. Performance evaluation and monitoring. **ACM Comput. Surv.**, Association for Computing Machinery, v. 3, n. 3, p. 79–91, 1971. ISSN 0360-0300. Citado nas páginas 45 e 46.

LUO, Q.; POSHYVANYK, D.; GRECHANIK, M. Mining performance regression inducing code changes in evolving software. In: **2016 IEEE/ACM 13th MSR**. [S.l.: s.n.], 2016. p. 25–36. Citado nas páginas 53, 55 e 56.

LUO, Q.; POSHYVANYK, D.; NAIR, A.; GRECHANIK, M. Forepost: A tool for detecting performance problems with feedback-driven learning software testing. In: **2016 IEEE/ACM 38th ICSE-C**. [S.l.: s.n.], 2016. p. 593–596. Citado nas páginas 53 e 55.

MÄKINEN, S.; SKOGSTRÖM, H.; LAAKSONEN, E.; MIKKONEN, T. Who needs mlops: What data scientists seek to accomplish and how can mlops help? **CoRR**, abs/2103.08942, 2021. Disponível em: <https://arxiv.org/abs/2103.08942>. Citado na página 40.

MARIE-SAINTE, S. L.; ALALYANI, N.; ALOTAIBI, S.; GHOUZALI, S.; ABUNADI, I. Arabic natural language processing and machine learning-based systems. **IEEE Access**, v. 7, p. 7011–7020, 2019. Citado na página 34.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **The bulletin of mathematical biophysics**, v. 5, n. 4, p. 115–133, Dec 1943. ISSN 1522-9602. Disponível em: <https://doi.org/10.1007/BF02478259>. Citado na página 33.

MCKINNEY, W. **Pandas Documentation**. 2021. Disponível em: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.corr.html>. Citado na página 101.

MESKINI, S.; NASSIF, A.; CAPRETZ, L. Reliability models applied to mobile applications. In: . [S.l.: s.n.], 2013. Citado na página 42.

MICROSOFT. **Overview of Windows Performance Monitor**. 2021. Disponível em: <https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-r2-and-2008/>. Citado nas páginas 48 e 72.

____. **Performance Counters**. 2021. Disponível em: <https://docs.microsoft.com/en-us/windows/win32/perfctrs/performance-counters-portal>. Citado na página 72.

MIHAYLOV, A.; CORVINELLI, V.; GODFREY, P.; MIERZEJEWSKI, P.; SZLICHTA, J.; ZUZARTE, C. Scalable learning to troubleshoot query performance problems. In: **Proceedings of the 30th ACM International Conference on Information and Knowledge Management**. New York, NY, USA: Association for Computing Machinery, 2021. (CIKM '21), p. 4016–4025. ISBN 9781450384469. Disponível em: <https://doi.org/10.1145/3459637.3481947>. Citado na página 53.

MITRA, S.; BRONEVETSKY, G.; JAVAGAL, S.; BAGCHI, S. Dealing with the unknown: Resilience to prediction errors. In: **2015 PACT**. [S.l.: s.n.], 2015. p. 331–342. Citado na página 53.

MOHRI, M.; ROSTAMIZADEH, A.; TALWALKAR, A. **Foundations of Machine Learning**. [S.l.]: The MIT Press, 2012. ISBN 026201825X. Citado nas páginas 22, 27, 31, 32 e 34.

MONRTENSEM, A. **GPUtil Documentation**. 2021. Disponível em: <https://github.com/anderskm/gputil>. Citado na página 74.

MONTES, V. S. **Detecção de defeitos de software utilizando agrupamento de perfis de desempenho**. Dissertação (Mestrado) — Universidade de São Paulo, 2019. Citado nas páginas 22, 23, 24, 25, 26, 49, 51, 57, 58, 59, 61, 63, 75, 95 e 99.

MUSTAFA, K. M.; AL-QUTAISH, R. E.; MUHAIRAT, M. I. Classification of software testing tools based on the software testing methods. In: **2009 Second International Conference on Computer and Electrical Engineering**. [S.l.: s.n.], 2009. v. 1, p. 229–233. Citado na página 44.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3rd. ed. [S.l.]: Wiley Publishing, 2011. ISBN 1118031962. Citado nas páginas 22, 23, 36, 39 e 40.

MüHLBAUER, S.; APEL, S.; SIEGMUND, N. Accurate modeling of performance histories for evolving software systems. In: **2019 34th IEEE/ACM ASE**. [S.l.: s.n.], 2019. p. 640–652. Citado na página 53.

NIKANJAM, A.; MOROVATI, M. M.; KHOMH, F.; BRAIEK, H. B. Faults in deep reinforcement learning programs: A taxonomy and A detection approach. **CoRR**, abs/2101.00135, 2021. Disponível em: <https://arxiv.org/abs/2101.00135>. Citado nas páginas 79, 80 e 81.

NILSSON, N.; NILSSON, N. **Artificial Intelligence: A New Synthesis**. Morgan Kaufmann Publishers, 1998. ISBN 9781558605350. Disponível em: <https://books.google.com.br/books?id=GYOFSd6fETgC>. Citado na página 28.

NVIDIA. **NvProf**. 2021. Disponível em: <https://docs.nvidia.com/cuda/profiler-users-guide/index.html>. Citado na página 73.

PANG, M. **Chess Alpha Zero**. 2020. Disponível em: <https://github.com/Zeta36/chess-alpha-zero>. Citado nas páginas 67 e 83.

PETERSEN, K.; VAKKALANKA, S.; KUZNIARZ, L. Guidelines for conducting systematic mapping studies in software engineering: An update. **Information and Software Technology**, v. 64, p. 1–18, 2015. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584915000646>. Citado na página 51.

POOLE, D.; MACKWORTH, A.; GOEBEL, R. **Computational Intelligence: A Logical Approach**. [S.l.: s.n.], 1998. ISBN 978-0-19-510270-3. Citado na página 28.

PRESSMAN, R. **Software Engineering: A Practitioner's Approach**. 7. ed. USA: McGraw-Hill, Inc., 2009. ISBN 0073375977. Citado na página 27.

PRESTON-WERNER CHRIS WANSTRATH, P. J. H. T.; CHACON, S. **GitHub**. 2021. Disponível em: <https://github.com/>. Citado nas páginas 66, 79 e 82.

PREUVENEERS, D.; TSINGENOPOULOS, I.; JOOSEN, W. Resource usage and performance trade-offs for machine learning models in smart environments. **Sensors**, v. 20, n. 4, 2020. ISSN 1424-8220. Disponível em: <https://www.mdpi.com/1424-8220/20/4/1176>. Citado na página 68.

RAJPURKAR, P. **SQuAD Benchmark**. 2021. Disponível em: <https://rajpurkar.github.io/SQuAD-explorer/>. Citado na página 85.

REICHELT, D. G.; KüHNE, S. How to detect performance changes in software history: Performance analysis of software system versions. In: . [S.l.]: Association for Computing Machinery, 2018. (ICPE '18), p. 183–188. ISBN 9781450356299. Citado na página 53.

REICHELT, D. G.; KüHNE, S.; HASSELBRING, W. Peass: A tool for identifying performance changes at code level. In: . [S.l.]: IEEE Press, 2019. (ASE '19), p. 1146–1149. ISBN 9781728125084. Citado na página 53.

REN, Y.; YOO, S.; HOISIE, A. Performance analysis of deep learning workloads on leading-edge systems. In: **2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)**. [S.l.: s.n.], 2019. p. 103–113. Citado na página 68.

RICH, E.; KNIGHT, K. **Artificial Intelligence**. 2nd. ed. [S.l.]: McGraw-Hill Higher Education, 1990. ISBN 0070522634. Citado na página 28.

RODOLA, G. **Psutil Documentation**. 2021. Disponível em: <https://psutil.readthedocs.io/en/latest/>. Citado nas páginas 48, 49 e 73.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. 3rd. ed. USA: Prentice Hall Press, 2009. ISBN 0136042597. Citado nas páginas 21, 27, 28, 29, 30, 33 e 34.

SAMUEL, A. L. Some studies in machine learning using the game of checkers. **IBM J. Res. Dev.**, v. 3, p. 210–229, 1959. Citado na página 21.

SANCHES, A.; CARDOSO, J. M. P.; DELBEM, A. C. B. Identifying merge-beneficial software kernels for hardware implementation. In: **2011 International Conference on Reconfigurable Computing and FPGAs**. [S.l.: s.n.], 2011. p. 74–79. Citado nas páginas 23, 32, 57 e 59.

SáNCHEZ, A. B.; DELGADO-PéREZ, P.; MEDINA-BULO, I.; SEGURA, S. Search-based mutation testing to improve performance tests. In: . [S.l.]: Association for Computing Machinery, 2018. (GECCO '18), p. 316–317. ISBN 9781450357647. Citado na página 53.

SANTOS, B. S. S. **Dataset - On Using Decision Tree Coverage Criteria for Testing Machine Learning Models**. 2021. Disponível em: <https://zenodo.org/record/4772950>. Citado na página 67.

SANTOS, S.; SILVEIRA, B.; DURELLI, V.; DURELLI, R.; SOUZA, S.; DELAMARO, M. On using decision tree coverage criteria fortesting machine learning models. In: ____. New York, NY, USA: Association for Computing Machinery, 2021. p. 1–9. ISBN 9781450385039. Disponível em: <https://doi.org/10.1145/3482909.3482911>. Citado na página 67.

SAUVANAUD, C.; SILVESTRE, G.; KAâNICHE, M.; KANOUN, K. Data stream clustering for online anomaly detection in cloud applications. In: **2015 11th EDCC**. [S.l.: s.n.], 2015. p. 120–131. Citado na página 53.

SCIENCE, W. of. **Analyze Results**. 2021. Disponível em: <https://www.webofscience.com/wos/woscc/analyze-results/>. Citado na página 66.

SHEN, D.; LUO, Q.; POSHYVANYK, D.; GRECHANIK, M. Automating performance bottleneck detection using search-based application profiling. In: . [S.l.]: Association for Computing Machinery, 2015. (ISSTA 2015), p. 270–281. ISBN 9781450336208. Citado nas páginas 53 e 81.

SHEN, Q.; MA, H.; CHEN, J.; TIAN, Y.; CHEUNG, S.-C.; CHEN, X. A comprehensive study of deep learning compiler bugs. In: **Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2021. (ESEC/FSE 2021), p. 968–980. ISBN 9781450385626. Disponível em: <https://doi.org/10.1145/3468264.3468591>. Citado na página 79.

SILVA, V.; NEVES, L.; SOUZA, R.; COUTINHO, A. L.; de Oliveira, D.; MATTOSO, M. Adding domain data to code profiling tools to debug workflow parallel execution. **Future Generation Computer Systems**, v. 110, p. 422–439, 2020. ISSN 0167-739X. Citado nas páginas 53 e 56.

SILVER, D.; HUBERT, T.; SCHRITTWIESER, J.; ANTONOGLOU, I.; LAI, M.; GUEZ, A.; LANCTOT, M.; SIFRE, L.; KUMARAN, D.; GRAEPEL, T.; LILLICRAP, T. P.; SIMONYAN, K.; HASSABIS, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. **CoRR**, abs/1712.01815, 2017. Disponível em: <http://arxiv.org/abs/1712.01815>. Citado na página 67.

SNEHA, K.; MALLE, G. M. Research on software testing techniques and software automation testing tools. In: **2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)**. [S.l.: s.n.], 2017. p. 77–81. Citado na página 44.

SOMMERVILLE, I. **Software Engineering**. 9th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0137035152. Citado nas páginas 37, 39, 41 e 42.

SOUSA, V. S.; NEVES, L.; SOUZA, R.; COUTINHO, A.; OLIVEIRA, D. de; MATTOSO, M. Integrating domain-data steering with code-profiling tools to debug data-intensive workflows. In: **WORKS@SC**. [S.l.: s.n.], 2016. Citado na página 53.

SOUZA, S.; MALDONADO, J.; PINTO, S.; FABBRI, F.; AURI, M.; VINCENZI, A.; BARBOSA, E.; DELAMARO, M.; JINO, M. IntroduÇÃo ao teste de software. In: ____. [S.l.: s.n.], 2000. p. 1–40. Citado nas páginas 22, 36, 38, 41 e 43.

SPOLSKY, J.; ATWOOD, J. **StackOverflow**. 2021. Disponível em: <https://stackoverflow.com/>. Citado na página 79.

SUJON, M.; SHAFIUZZAMAN, M.; RAHMAN, M. M.; RAHMAN, R. Characterization and localization of performance-bugs using naive bayes approach. In: **2016 5th ICIEV**. [S.l.: s.n.], 2016. p. 791–796. Citado na página 53.

SUN, X.; ZHOU, T.; LI, G.; HU, J.; YANG, H.; LI, B. An empirical study on real bugs for machine learning programs. In: **2017 24th Asia-Pacific Software Engineering Conference (APSEC)**. [S.l.: s.n.], 2017. p. 348–357. Citado nas páginas 79 e 81.

SáNCHEZ, A. B.; DELGADO-PéREZ, P.; MEDINA-BULO, I.; SEGURA, S. Tandem: A taxonomy and a dataset of real-world performance bugs. **IEEE Access**, v. 8, p. 107214–107228, 2020. ISSN 2169-3536. Citado nas páginas 53 e 54.

SáNCHEZ, A. B.; DELGADO-PéREZ, P.; SEGURA, S.; MEDINA-BULO, I. Performance mutation testing: Hypothesis and open questions. **Information and Software Technology**, v. 103, p. 159–161, 2018. ISSN 0950-5849. Citado na página 53.

TAKEI, G. **George Takei Ted Talk**. 2021. Disponível em: <https://github.com/audio-samples/audio-samples.github.io/tree/master/samples/mp3/ted_speakers/GeorgeTakei>. Citado na página 90.

THUNG, F.; WANG, S.; LO, D.; JIANG, L. An empirical study of bugs in machine learning systems. In: **2012 IEEE 23rd International Symposium on Software Reliability Engineering**. [S.l.: s.n.], 2012. p. 271–280. Citado nas páginas 79 e 81.

TIZPAZ-NIARI, S.; CERNY, P.; TRIVEDI, A. Detecting and understanding real-world differential performance bugs in machine learning libraries. In: . [S.l.]: Association for Computing Machinery, 2020. (ISSTA 2020), p. 189–199. ISBN 9781450380089. Citado nas páginas 53, 55 e 56.

____. Detecting and understanding real-world differential performance bugs in machine learning libraries. In: . [S.l.]: Association for Computing Machinery, 2020. (ISSTA 2020), p. 189–199. ISBN 9781450380089. Citado na página 68.

USP. **USP ICMC**. 2021. Disponível em: <https://www.icmc.usp.br/pos-graduacao>. Citado na página 67.

VLADISHEV, A. **Zabbix**. 2021. Disponível em: <https://www.zabbix.com/>. Citado nas páginas 48, 49 e 71.

WANG, A. **GLUE Benchmark**. 2021. Disponível em: <https://gluebenchmark.com/>. Citado na página 85.

WANG, M.; MENG, C.; LONG, G.; WU, C.; YANG, J.; LIN, W.; JIA, Y. Characterizing deep learning training workloads on alibaba-pai. In: . [S.l.: s.n.], 2019. p. 189–202. Citado na página 68.

WANG, T.; ZHANG, W.; WEI, J.; ZHONG, H. Fault detection for cloud computing systems with correlation analysis. In: **2015 IFIP/IEEE IM**. [S.l.: s.n.], 2015. p. 652–658. Citado nas páginas 53 e 56.

WANG, X.; JIN, Y.; YU, Y. A mobile network performance evaluation method based on multivariate time series clustering with auto-encoder. In: . [S.l.]: Association for Computing Machinery, 2018. (ICTCE 2018), p. 33–37. ISBN 9781450365857. Citado na página 53.

WANG, Y.; WU, Z.; LI, Q.; ZHU, Y. A model of telecommunication network performance anomaly detection based on service features clustering. **IEEE Access**, v. 5, p. 17589–17596, 2017. Citado nas páginas 53 e 55.

WEN, Z.; DAI, W.; ZOU, D.; JIN, H. Perfdoc: Automatic performance bug diagnosis in production cloud computing infrastructures. In: **2016 IEEE Trustcom/BigDataSE/ISPA**. [S.l.: s.n.], 2016. p. 683–690. ISSN 2324-9013. Citado na página 53.

WERT, A.; SCHULZ, H.; HEGER, C. Aim: Adaptable instrumentation and monitoring for automated software performance analysis. In: **2015 IEEE/ACM 10th AST**. [S.l.: s.n.], 2015. p. 38–42. Citado na página 53.

WIENKE, J.; WREDE, S. Autonomous fault detection for performance bugs in component-based robotic systems. In: **2016 IEEE/RSJ IROS**. [S.l.: s.n.], 2016. p. 3291–3297. Citado na página 53.

WINSTON, P. H. **Artificial Intelligence (3rd Ed.)**. USA: Addison-Wesley Longman Publishing Co., Inc., 1992. ISBN 0201533774. Citado na página 28.

WU, D.; SHEN, B.; CHEN, Y. An empirical study on tensor shape faults in deep learning systems. **CoRR**, abs/2106.02887, 2021. Disponível em: <https://arxiv.org/abs/2106.02887>. Citado na página 79.

WURZENBERGER, M.; SKOPIK, F.; FIEDLER, R.; KASTNER, W. Applying high-performance bioinformatics tools for outlier detection in log data. In: **2017 3rd IEEE CYBCONF**. [S.l.: s.n.], 2017. p. 1–8. Citado na página 53.

ZHANG, S.; LIU, D.; ZHOU, L.; REN, Z.; WANG, Z. Diagnostic framework for distributed application performance anomaly based on adaptive instrumentation. In: **2020 2nd ICCCI**. [S.l.: s.n.], 2020. p. 164–169. Citado na página 53.

ZHANG, X.; WANG, Y.; SHI, W. pcamp: Performance comparison of machine learning packages on the edges. In: **USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)**. Boston, MA: USENIX Association, 2018. Disponível em: <https://www.usenix.org/conference/hotedge18/presentation/zhang>. Citado na página 68.

ZHANG, Y.; CHEN, Y.; CHEUNG, S.-C.; XIONG, Y.; ZHANG, L. An empirical study on tensorflow program bugs. In: **Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis**. New York, NY, USA: Association for Computing Machinery, 2018. (ISSTA 2018), p. 129–140. ISBN 9781450356992. Disponível em: <https://doi.org/10.1145/3213846.3213866>. Citado nas páginas 79, 80 e 81.

ZINS, P.; DAGENAIS, M. Tracing and profiling machine learning dataflow applications on gpu. **Int J Parallel**, 2019. Citado na página 68.


ICMC
USP
SÃO CARLOS